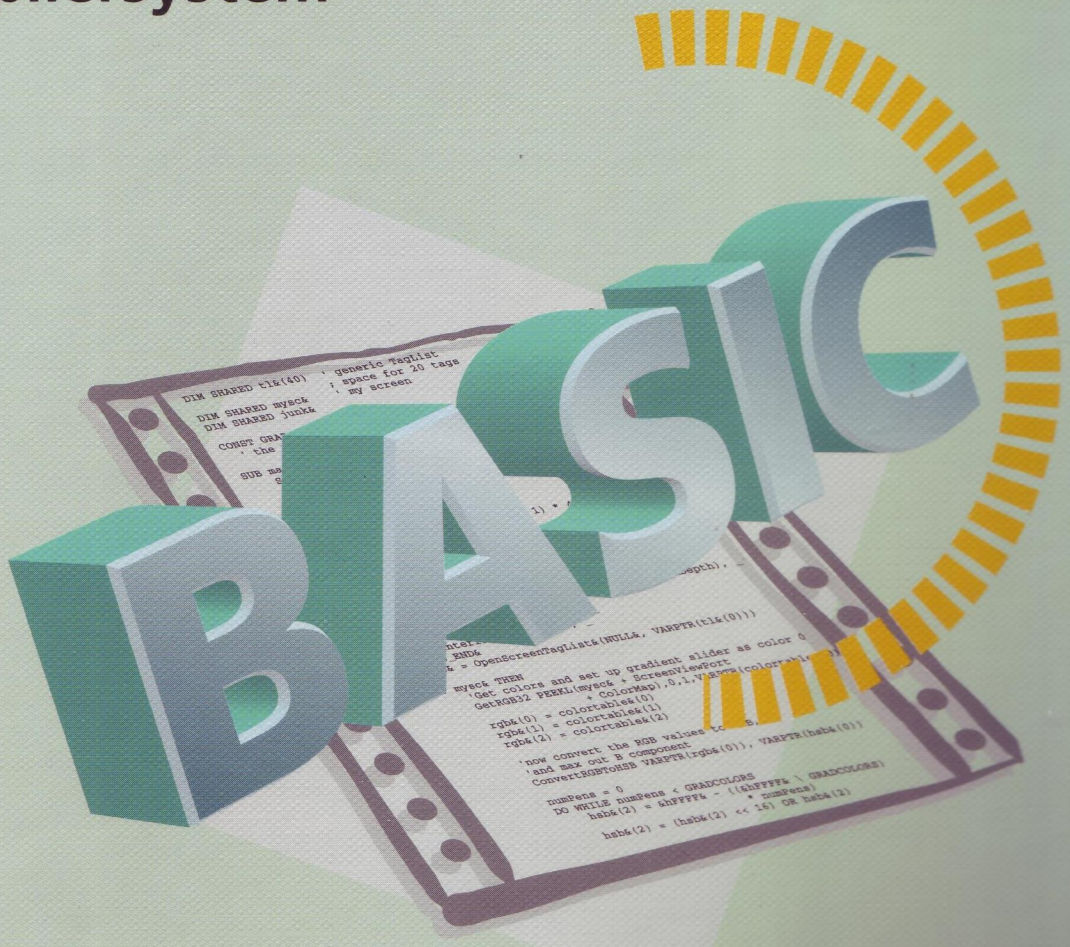
Maxon BASIC 3

Integriertes Basic-Compilersystem



AMIGA®

MAXON computer

AMIGA

MaxonBASIC 3

Das professionelle BASIC-System der 90er

MAXON
computer

MaxonBASIC 3

© 1994 by MAXON Computer GmbH, HiSoft

Alle Rechte vorbehalten. Dieses Handbuch und die dazugehörige Software ist urheberrechtlich geschützt. Es darf in keiner Form (auch auszugsweise) mittels irgendwelcher Verfahren reproduziert, gesendet, vervielfältigt bzw. verbreitet oder in eine andere Sprache übersetzt werden.

Bei der Erstellung des Programms, der Anleitung sowie Abbildungen wurde mit allergrößter Sorgfalt vorgegangen. Trotzdem können Fehler nicht ausgeschlossen werden. MAXON übernimmt keinerlei Haftung für Schäden, die auf eine Fehlfunktion von Programmen zurückzuführen sind.

Alle Informationen, die in der vorliegenden Anleitungen enthalten sind, werden ohne Rücksicht auf einen eventuellen Patentschutz veröffentlicht. Ebenso werden Warenzeichen ohne Gewährleistung einer freien Verwendung benutzt.

Support

Bei Fragen zu **Maxon**BASIC wenden Sie sich bitte an:

MAXON Computer
"**Maxon**BASIC - Support"
Industriestr. 26
D-65760 Eschborn

Copyrights und Warenzeichen:
Commodore® und Amiga® sind eingetragene Warenzeichen der Commodore-Amiga Inc.
AmigaDOS™, Kickstart™, Intuition™ und Workbench™ sind Warenzeichen der Commodore-Amiga Inc.
Amiga Workbench™ © Commodore-Amiga Inc.
MAXON® ist ein eingetragenes Warenzeichen der MAXON Computer GmbH

MAXON Software-Lizenzbedingungen

1 Allgemeines

Gegenstand dieses Vertrages ist das Benutzungsrecht für MAXON Computerprogramme, für die Benutzungsanleitung sowie für sonstiges zugehöriges schriftliches Material, nachfolgend zusammenfassend als Software bezeichnet.

2 Vervielfältigungsrechte

(1) Der Anwender darf das gelieferte Programm vervielfältigen, soweit die jeweilige Vervielfältigung für die Benutzung des Programms notwendig ist. Zu den notwendigen Vervielfältigungen zählen die Installation des Programms vom Originaldatenträger auf den Massenspeicher der eingesetzten Hardware sowie das Laden des Programms in den Arbeitsspeicher.

(2) Darüber hinaus kann der Anwender eine Vervielfältigung zu Sicherungszwecken vornehmen. Es darf jedoch jeweils nur eine einzige Sicherungskopie angefertigt und aufbewahrt werden. Diese Sicherungskopie ist als solche des überlassenen Programms zu kennzeichnen.

(3) Weitere Vervielfältigungen, zu denen auch die Ausgabe des Programmcodes auf einen Drucker sowie das Fotokopieren des Handbuchs zählen, darf der Anwender nicht anfertigen.

3 Mehrfachnutzungen und Netzwerkeinsatz

(1) Der Anwender darf die Software auf jeder ihm zur Verfügung stehenden Hardware einsetzen. Wechselt der Anwender jedoch die Hardware, muß er die Software vom Massenspeicher der bisher verwendeten Hardware löschen. Ein zeitgleiches Einspeichern, Vorrätighalten oder Benutzen auf mehr als nur einer Hardware ist unzulässig.

(2) Der Einsatz der überlassenen Software innerhalb eines Netzwerkes oder eines sonstigen Mehrstations-Rechensystems ist unzulässig, sofern damit die Möglichkeit zeitgleicher Mehrfachnutzung des Programms geschaffen wird. Möchte der Anwender die Software innerhalb eines Netzwerkes oder sonstiger Mehrstations-Rechensysteme einsetzen, muß er eine zeitgleiche Mehrfachnutzung durch Zugriffsschutzmechanismen unterbinden oder an den Lieferanten eine besondere Netzwerkgebühr entrichten, deren Höhe sich nach der Anzahl der an das Rechensystem angeschlossenen Benutzer bestimmt.

(3) Die im Einzelfall zu entrichtende Netzwerkgebühr wird dem Anwender von dem Lieferant umgehend mitgeteilt, sobald der Anwender dem Lieferanten den geplanten Netzwerkeinsatz einschließlich der Anzahl angeschlossener Benutzer schriftlich bekanntgegeben hat. Unsere Anschrift ist dem Benutzerhandbuch zu entnehmen. Der Einsatz im Netzwerk ist erst nach der vollständigen Entrichtung der Netzwerkgebühr zulässig.

4 Weiterveräußerung

(1) Der Anwender darf die Software einschließlich des Benutzerhandbuchs und des sonstigen Begleitmaterials auf Dauer an Dritte veräußern oder verschenken, vorausgesetzt, der erwerbende Dritte erklärt sich mit der Weitergeltung der vorliegenden Vertragsbedingungen dem Anwender gegenüber einverstanden.

(2) Der Anwender muß die vorliegenden Vertragsbedingungen sorgfältig aufbewahren. Vor der Weitergabe der Software muß er sie dem neuen Anwender zur Kenntnisnahme vorlegen. Sollte der Anwender zum Zeitpunkt der Weitergabe die vorliegenden Vertragsbedingungen nicht mehr im Besitz haben, ist er verpflichtet, ein Ersatzexemplar beim Hersteller anzufordern. Die entstehenden Versandkosten trägt der Anwender.

(3) Im Falle der Weitergabe muß der Anwender dem neuen Anwender sämtliche Programmkopien einschließlich gegebenenfalls vorhandener Sicherheitskopien übergeben oder die nicht übergebenen Kopien vernichten. Infolge der Weitergabe erlischt das Recht des alten Anwenders zur Programmnutzung.

Der Anwender ist verpflichtet, der Informationspflicht nach 11 dieses Vertrages nachzukommen.

5 Rekompilierung und Programmänderungen

(1) Die Rückübersetzung des überlassenen Programmcodes in andere Codeformen (Rekompilierung) sowie sonstige Arten der Rückerschließung der verschiedenen Herstellungsstufen der Software (Reverse-Engineering) einschließlich einer Programmänderung sind im privaten Bereich zulässig.

(2) Die Entfernung eines Kopierschutzes oder ähnlicher Schutzroutinen ist nur zulässig, sofern durch diesen Schutzmechanismus die störungsfreie Pogrammnutzung beeinträchtigt oder verhindert wurde. Für die Beeinträchtigung oder Verhinderung störungsfreier Benutzbarkeit durch den Schutzmechanismus trägt der Anwender die Beweislast.

(3) Die in den voranstehenden beiden Absätzen angesprochenen Handlungen dürfen nur dann kommerziell arbeitenden Dritten überlassen werden, die mit dem Lieferanten in einem potentiellen Wettbewerbsverhältnis stehen, wenn der Lieferant die gewünschten Programmänderungen nicht gegen ein angemessenes Entgelt vornehmen will. Der Anwender muß dem Lieferanten eine hinreichende Frist zur Prüfung der Auftragsübernahme einräumen.

(4) Urherberrechtsvermerke, Seriennummern sowie sonstige der Programmidentifikation dienende Merkmale dürfen auf keinen Fall entfernt oder verändert werden.

6 Gewährleistung

(1) Mängel der gelieferten Software einschließlich der Handbücher und sonstiger Unterlagen werden vom Lieferanten innerhalb der Gewährleistungsfrist von sechs Monaten ab Lieferung nach entsprechender Mitteilung durch den Anwender

behoben. Dies geschieht nach Wahl des Lieferanten durch kostenfreie Nachbesserung oder durch Ersatzlieferung in Form eines Updates.

(2) Kann der Mangel nicht innerhalb angemessener Frist behoben werden, oder ist die Nachbesserung oder Ersatzlieferung aus sonstigen Gründen als fehlgeschlagen anzusehen, kann der Anwender nach seiner Wahl Herabsetzung der Vergütung (Minderung) oder Rückgängigmachung des Vertrages (Wandelung) verlangen. Von einem Fehlschlagen der Nachbesserung oder der Ersatzlieferung ist erst auszugehen: wenn dem Lieferanten hinreichende Gelegenheit zur Nachbesserung oder Ersatzlieferung eingeräumt wurde, ohne daß der gewünschte Erfolg erzielt wurde; wenn die Nachbesserung oder Ersatzlieferung unmöglich ist; wenn sie vom Lieferanten verweigert oder unzumutbar verzögert wird; wenn begründete Zweifel hinsichtlich der Erfolgsaussichten bestehen; oder wenn eine Unzumutbarkeit aus sonstigen Gründen vorliegt.

(3) Der Anwender weiß, daß nach dem heutigen Stand der Technik die Erstellung völlig fehlerfreier Software nicht möglich ist.

7 Untersuchungs- und Rügepflicht

(1) Der Anwender ist verpflichtet, die gelieferte Software auf offensichtliche Mängel, die einem durchschnittlichen Kunden ohne weiteres auffallen, zu untersuchen. Offensichtliche Mängel, insbesondere das Fehlen von Datenträgern oder Handbüchern sowie erhebliche, leicht sichtbare Beschädigungen des Datenträgers, sind beim Lieferanten innerhalb von zwei Wochen nach Lieferung schriftlich zu rügen. Die Mängel, insbesondere die aufgetretenen Symptome, sind nach Kräften detailliert zu beschreiben.

(2) Mängel, die nicht offensichtlich sind, müssen beim Lieferanten innerhalb von zwei Wochen nach dem Erkennen durch den Anwender gerügt werden.

(3) Bei Verletzung der Untersuchungs- und Rügepflicht gilt die Software in Ansehung des betreffenden Mangels als genehmigt.

8 Haftung

(1) Für Schäden wegen Rechtsmängeln und Fehlens zugesicherter Eigenschaften haftet der Lieferant unbeschränkt. Die Haftung für anfängliches Unvermögen, Verzug und Unmöglichkeit wird auf das Fünffache des Überlassungsentgelts sowie auf solche Schäden begrenzt, mit deren Entstehung im Rahmen einer allgemeinen Software-Überlassung typischerweise gerechnet werden muß. Im übrigen haftet der Lieferant nur für Vorsatz und grobe Fahrlässigkeit auch seiner gesetzlichen Vertreter und Erfüllungsgehilfen, sofern nicht eine Pflicht verletzt wird, deren Einhaltung für die Erreichung des Vertragszwecks von besonderer Bedeutung ist (Kardinalpflicht).

Bei Verletzung der Kardinalpflicht ist die Haftungsbeschränkung für anfängliches Unvermögen entsprechend heranzuziehen.

(2) Der Lieferant haftet nicht für Schäden, die durch Fehlbenutzung der Rechenanlage oder durch mangelnde, regelmäßige Absicherung der Daten in Form von Sicherungskopien entstanden sind.

9 Eigentumsvorbehalt

(1) Der Lieferant behält sich das Eigentum an der dem Anwender gelieferten Software bis zur vollständigen Bezahlung sämtlicher zum Zeitpunkt der Lieferung bestehender oder später entstehender Forderungen aus diesem Vertragsverhältnis vor; bei Bezahlung durch Scheck oder Wechsel bis zu deren Einlösung.

(2) Bei verschuldeten Zahlungsrückständen des Anwenders gilt die Geltendmachung des Eigentumsvorbehalts durch den Lieferanten nicht als Rücktritt vom Vertrag, es sei denn, der Lieferant teilt dies dem Anwender ausdrücklich mit.

10 Transportschäden

Der Anwender ist verpflichtet, eventuelle Transportschäden unverzüglich und schriftlich dem Transporteur zu melden und dem Lieferanten eine Kopie des Schriftverkehrs zuzuschicken, denn alle Sendungen sind über den Lieferanten versichert.

11 Informationspflichten

Der Anwender ist im Falle der Weiterveräußerung der Software verpflichtet, dem Lieferanten den Namen und die vollständige Anschrift des Käufers schriftlich mitzuteilen. Die Adresse des Lieferanten ist der vorderen Umschlagseite der Benutzungsanleitung zu entnehmen.

12 Sonstiges

(1) In diesem Vertrag sind sämtliche Rechte und Pflichten der Vertragsparteien geregelt. Sonstige Vereinbarungen bestehen nicht. Änderungen sind nur in Schriftform und bei Bezugnahme auf diesen Vertrag wirksam und beiderseitig zu unterzeichnen.

(2) Der Gerichtsstand für alle Streitigkeiten aus diesem Vertrag ist, soweit vereinbar, das zuständige Gericht in Frankfurt am Main.

(3) Sollten einzelne Bestimmungen dieser Bedingungen nicht rechtswirksam sein oder ihre Rechtswirksamkeit durch einen späteren Umstand verlieren, oder sollte sich in diesen Bedingungen eine Lücke herausstellen, so wird hierdurch die Rechtswirksamkeit der übrigen Bestimmungen nicht berührt. Anstelle der unwirksamen Vertragsbestimmungen oder zur Ausfüllung der Lücke soll eine angemessene Regelung gelten, die, soweit rechtlich möglich, dem am nächsten kommt, was die Vertragsparteien gewollt haben, soweit sie von der Unwirksamkeit der Bestimmung Kenntnis gehabt hätten.

Inhaltsverzeichnis

Einführung **11**
Befehlsübersicht **12**
ABS Funktion 13
AREA Befehl 14
AREAFILL Befehl 15
ASC Funktion 16
ATN Funktion 17
BEEP Befehl 18
BEGINIO Befehl 19
BIN$ Funktion 20
BLOAD Befehl 21
BREAK Befehl 22
BSAVE Befehl 23
CALL Befehl 24
CALL LOC Befehl 26
CALLS Befehl 27
COMMAND$ Funktion 45
COMMON SHARED Befehl 46
CONST Befehl 47
COS Funktion 48
CSNG Funktion 49
CSRLIN Funktion 50
CURDIR$ Funktion 51
DATE$ Funktion 56
DECLARE statement 57
DECR Befehl 59
DEF FN Befehl 60
DEFDBL, DEFINT, DEFLNG, DEFSNG, DEFSTR Befehle 63
DIM Befehl 64
DO...LOOP Befehl 66
END Befehl 68
EOF Funktion 69
ERASE Befehl 70
ERL, ERR Funktions 71
ERROR Befehl 72
EXIT Befehl 73

EXP Funktion 74
FEXISTS Funktion 75
FILES Befehl 77
FIX Funktion 78
FOR...NEXT Befehl 79
FORMATI$, FORMATL$,
FORMATS$ und FORMATD$ Funktionen 81
FRE Funktion 82
FREEFILE Funktion 83
FUNCTION...END FUNCTION Befehl 84
GET Dateiein-/ausgabebefehl 86
GET Grafikbefehl 87
GOSUB...RETURN Befehl 88
GOTO Befehl 90
HEX$ Funktion 91
IF...THEN...ELSE Befehl 92
INCR Befehl 94
INITHOOK Befehl 95
INKEY$ Funktion 96
INPUT Befehl 98
INPUT# Befehl 99
INPUT$ Befehl 100
INSTR Funktion 101
INT Funktion 102
KILL Befehl 103
LBOUND Funktion 104
LCASE$ Funktion 105
LEFT$ Funktion 106
LEN Funktion 107
LET Befehl 108
LIBRARY Befehl 109
LIBRARY Funktion 113
LINE Befehl 114
LINE INPUT Befehl 116
LINE INPUT# Befehl 117
LOC Funktion 118
LOCAL Befehl 119
LOCATE Befehl 120
LOF Funktion 121
LOG, LOG10, LOG2 Funktionen 122

LPOS Funktion 123
LPRINT, LPRINT USING Befehle 124
LSET Befehl 125
LTRIM$ Funktion 126
MAX Funktion 127
MENU Befehl 128
MENU Funktion 130
MID$ Funktion 131
MID$ Befehl 132
MIN Funktion 133
MKDIR Befehl 134
MKI$, MKFFP$, MKL$, MKS$, MKD$ Funktionen 135
MOUSE Funktion 136
MOUSE Befehl 138
NAME Befehl 139
OBJECT Befehle & Funktionen 140
OCT$ function 144
ON...BREAK Befehl 145
ON...CLOSE Befehl 146
ON...COLLISION Befehl 147
ON...ERROR Befehl 148
ON...GOSUB Befehl 150
ON...GOTO Befehl 151
ON...MENU Befehl 152
ON...MOUSE Befehl 154
ON...TIMER Befehl 155
OPEN Befehl 156
OPTION BASE Befehl 160
PAINT Befehl 161
PALETTE Befehl 162
PATTERN Befehl 163
PCOPY Befehl 164
PEEK, PEEKB, PEEKL, PEEKW, PEEK$ Funktionen 165
POINT Funktion 167
POKE, POKEB, POKEL, POKEW Befehle 168
POS Funktion 169
PRESET Befehl 170
PRINT Befehl 171
PRINT#, PRINT# USING

Befehle 173
PRINT USING Befehl 174
PSET Befehl 177
PTAB Funktion 178
PUT Ein-/Ausgabebefehl 179
PUT Grafikbefehl 180
RANDOMIZE Befehl 182
READ Befehl 183
REDIM Befehl 184
REM Befehl 185
REPEAT...END REPEAT Befehl 186
RESET Befehl 188
RESTORE Befehl 189
RESUME Befehl 190
RETURN Befehl 191
RIGHT$ Funktion 192
RINSTR Funktion 193
RMDIR Befehl 194
RUN Befehl 198
SADD Funktion 199
SAY Befehl 200
SCREEN Befehl 201
SCROLL Befehl 203
SELECT...END SELECT Befehl 204
SGN Funktion 206
SHARED Befehl 207
SIN Funktion 208
SLEEP Befehl 209
SOUND Befehl 210
SPACE$ Funktion 212
SPC Funktion 213
SQR Funktion 214
STATIC Befehl 215
STICK Funktion 216
STOP Befehl 217
STR$ Funktion 218
STRIG Funktion 219
STRING$ Funktion 220
SUB...END SUB Befehl 221
SWAP Befehl 223

SYSTAB Funktion 224
TAB Funktion 226
TAGLIST Befehl 227
TAN Funktion 229
TIME$ Funktion 230
TIMER Befehl 231
TIMER Funktion 232
TRANSLATE$ Funktion 233
TRON,TROFF Befehle 234
UBOUND Funktion 235
UCASE$ Funktion 236
VAL Funktion 237
VARPTR Funktion 238
VARPTRS Funktion 239
WAVE Befehl 240
WHILE...WEND Befehl 241
WIDTH Befehl 242
WINDOW Befehle 243
WINDOW Funktion 246
WRITE Befehl 247
WRITE# Befehl 248
Tutorial ***249***
Das erste Programm 250
Das Aussehen eines BASIC-Befehls 250
Das Aussehen von BASIC-Programmen 251
Variablen - in ständiger Veränderung 251
Die x-te Wiederholung 252
Entscheidungen 253
Logische Abfragen 254
Der Editor ***256***
Starten des Editors 257
Bedienung des Editors 257
Die Menüs des Editors 257
Der Compiler ***265***
Das Compilieren 267
Compileroptionen 267
Menüs der Compiler-Optionen 274
Benutzung des CLI-Compilers HBC 274
Erstellung von CLI-Programmen 275
Tokenisieren 275

Einführung

MaxonBASIC (im folgenden auch MB) ist ein BASIC-Compiler, der sich wie ein Interpreter bedienen läßt. Der Vorteil ist die hohe Geschwindigkeit der damit erstellten Programme, so daß man MaxonBASIC für alle Aufgaben einsetzen kann.
Dem Einsteiger bietet MaxonBASIC ein einfach zu bedienendes System mit vielen BASIC-Befehlen, die die Programmierung sehr erleichtern.

Professionelle Programmierer finden in MaxonBASIC ein sehr schnelles und komfortables Entwicklungssystem. Besonders durch die neuen BASIC-Includes ist es sehr einfach möglich, Programme für die aktuelle Version 3.1 des AMIGA-Betriebssystems zu erstellen.

Damit Sie bei der Arbeit mit MaxonBASIC schnell Informationen zu Befehlen nachschlagen können, beginnt dieses Handbuch mit der Befehlsübersicht. Sie sollten aber dennoch zunächst das Tutorial ab Seite 249 und die folgenden Kaptitel lesen, um mit der Bedienung von MaxonBASIC vertraut zu werden.

Denken Sie unbedingt auch daran, die beiliegende Registrierkarte auszufüllen und an uns zurückzusenden. Nur so erhalten Sie bei uns Support und werden über neue Entwicklungen auf dem Laufenden gehalten.

Wir wünschen Ihnen nun viel Spaß bei der Programmierung mit MaxonBASIC. Vielleicht schicken Sie uns ja einmal ein von Ihnen erstelltes Programm. Wir freuen uns über jede Zusendung!

Befehlsübersicht

Dieses Kapitel gibt Ihnen eine ausführliche Beschreibung jeder Funktion und jedes Befehls von MaxonBASIC.
Dabei wird folgendermaßen vorgegangen:

• Syntax

Hier finden Sie, wie die Befehle und Funktionen zu gebrauchen sind. Parameter sind *kursiv* aufgeführt und – sofern optional – in eckigen Klammern geschrieben [ ].

• Wirkungsweise

Beschreibt kurz die Wirkungsweise des Befehls oder der Funktion.

• Details

Geht mit der Beschreibung, wo es notwendig erscheint, mehr in die Tiefe.

• Siehe auch

Hier wird auf andere Befehle verwiesen, die in irgendeiner Verbindung zu diesem Befehl stehen. Es ist zu empfehlen, sich auch die anderen Befehle anzusehen, um weitere Informationen zu bekommen.

• Beispiel

Zeigt eines oder mehrere Beispiele zu dem Befehl.

ABS Funktion

• Syntax

`ABS(numerischer Ausdruck)`

• Wirkungsweise

Diese Funktion gibt den Absolutwert eines numerischen Ausdrucks zurück.

• Details

Die Absolutwertfunktion gibt den Wert des *numerischen Ausdrucks* ohne Vorzeichen zurück. Sowohl der Absolutwert von –1 als auch der von 1 haben den gleichen Absolutwert, nämlich 1.

• Siehe auch

`SGN`

• Beispiel

```
PRINT ABS(6*(7)), ABS(6*(-7))
```

Ergebnis:

```
42 42
```

AREA Befehl

• Syntax

AREA [STEP] (*x,y*)

• Wirkungsweise

Fügt einen Punkt in die Liste ein, die von dem nächsten AREAFILL-Befehl benutzt wird.

• Details

Um ein gefülltes Polygon mit MaxonBASIC zu zeichnen, muß man die Eckpunkte des Polygons mittels des AREA-Befehls angeben, und anschließend das Polygon mit dem AREAFILL-Befehl zeichnen lassen.
Wenn STEP benutzt wird, beziehen sich die Koordinaten *x* and *y* relativ auf die aktuelle Position des schon definierten Zeichenpunktes, anderenfalls beziehen sie sich relativ auf die aktuelle Fensterpostion.
Maximal 20 AREA-Befehle können zwischen AREAFILL-Befehlen definiert werden; defineren Sie mehr, werden sie ignoriert. Es ist nicht notwendig, den Startpunkt wieder als Endpunkt zu definieren.
Wenn Sie AREAFILL benutzen, ohne irgendwelche Zeichenpunkte definiert zu haben, werden die **AreaInfo- und TmpRas-Strukturen** dieses Fensters alloziert. Das kann ganz nützlich sein, wenn Sie direkt mit den Betriebssystemfunktionen zeichnen wollen.
Um ein ungefülltes Polygon zu zeichnen, benutzen Sie den LINE-Befehl.

• Siehe auch

AREAFILL, LINE.

• Beispiel

```
AREA    (50,50)
AREA    STEP (40,0)
AREA    STEP (0,30)
AREAFILL
```

Ergebnis:
Zeichnet ein gefülltes Dreieck mit den Eckpunkten (50,50), (90,50), (90,80) in das aktuelle Fenster.

AREAFILL Befehl

• Syntax

AREAFILL [*Füllmodus*]

• Wirkungsweise

Füllt oder invertiert ein Polygon, das zuvor durch AREA-Befehle definiert wurde.

• Details

Füllmodus ist ein Integer-Ausdruck. Hat er den Wert

0 wird das Füllmuster, das zuvor mit dem PATTERN-Befehl definiert wurde, benutzt.

1 wird der Flächeninhalt invertiert.

Wird der *Füllmodus* weggelassen, wird die Fläche mit dem aktuellen Füllmuster gefüllt, als wenn AREAFILL 0 benutzt worden wäre. Sehen Sie unter dem AREA-Befehl nach, wie die einzelnen Zeichenpunkte gesetzt werden.

• Siehe auch

AREA, PATTERN

• Beispiel

```
AREA (20,20)
AREA (30,20)
AREA (60,30)
AREA (20,30)
AREAFILL 1
```

Ergebnis:
Invertiert eine keilförmige Fläche auf dem Bildschirm.

ASC Funktion

• Syntax

ASC(*String*)

• Wirkungsweise

Diese Funktion gibt den ASCII-Code des ersten Zeichens von *String* zurück.

• Details

Wenn der String ein Leer-String ist, erhält man einen "Illegal function call"-Fehler. Das Ergebnis ist eine Integer-Zahl.

• Siehe auch

CHR$

• Beispiel

```
X$="FORTSCHRITT"
PRINT ASC(X$)     'Gibt den ASCII-Code für "F" aus
```

Ergebnis:

```
70
```

ATN Funktion

• Syntax

ATN(*numerischer Ausdruck*)

• Wirkungsweise

Diese Funktion gibt den Arcustangens des *numerischen Ausdrucks* oder den Winkel, dessen Tangens dem *numerischen Ausdruck* entspricht, zurück.

• Details

Das Ergebnis wird im Bogenmaß im Bereich von $-\pi/2$ bis $\pi/2$ zurückgegeben.
Der *numerische Ausdruck* kann eine beliebige Zahl sein. ATN arbeitet normalerweise mit einfacher Genauigkeit; sollte die Zahl jedoch über eine doppelte Genauigkeit verfügen, gibt ATN auch eine doppelt genaue Zahl zurück.

• Siehe auch

SIN, TAN, COS.

• Beispiel

```
PRINT 4*ATN(1);4*ATN(1#)
```

Ergebnis:

```
 3.141593   3.141592653589794
```

BEEP Befehl

• Syntax

```
BEEP
```

• Wirkungsweise

Läßt den Bildschirm aufblinken.

• Details

BEEP benutzt das ASCII-Klingel-Zeichen; den selben Effekt kann man mit PRINT CHR$(7) erreichen;. Üblicherweise wird von BEEP kein Ton ausgegeben, sondern der Bildschirm blitzt kurz auf. Ab OS 2.0 können Sie in den Sound-Preferences wählen, ob zusätzlich zum Aufblitzen des Bildschirms ein Ton abgespielt werden soll.
Damit der BEEP-Befehl des Compilers unabhängig von der Systemeinstellung einen Ton ausgibt, muß man folgende Zeile ins Programm nehmen:

```
BEEP : SOUND 880,2
```

• Siehe auch

```
SOUND
```

• Beispiel

```
PRINT "Achtung!"
BEEP
```

BEGINIO Befehl

• Syntax

`BEGINIO` *`Adresse`*

• Wirkungsweise

Dieser Befehl bereitet eine asynchrone Ein-/Ausgabe vor, indem er den BEGINIO-Vektor eines Gerätes direkt aufruft.

• Details

Die *Adresse* ist ein Langwort und die Adresse einer **IORequest**-Struktur, die zuvor mit **OpenDevice** initialisiert und geöffnet wurde. Der Befehl arbeitet wie die **SendIO**-Funktion der **Exec-Library**, löscht allerdings nicht zuvor das **io_Flags**-Feld. Das ist nur bei einigen Geräten wie z.B. beim Audio-Device notwendig.
Dieser Befehl wurde in den Compiler aufgrund seiner ungewöhnlichen Register-Erfordernisse eingebaut.

• Siehe auch

"AUDIO.BAS"-Beispiel.

BIN$ Funktion

• Syntax

BIN$(*numerischer Ausdruck*)

• Wirkungsweise

Dieser Befehl gibt einen String zurück, der dem binären Wert eines *numerischen Ausdrucks* entspricht. Wenn der Ausdruck eine Integer-Zahl ist, ist der Ergebnis-String bis zu 16 Zeichen lang. Ist der Ausdruck allerdings eine Long-Integer-Zahl, kann der Ergebnis-String bis zu 32 Zeichen lang sein.

• Details

Der *numerische Ausdruck* wird gerundet, bevor er mit BIN$ umgerechnet wird. Wenn der Parameter eine Integer-Zahl ist, kann der Ergebnis-String 1 bis 4 Zeichen enthalten. Wenn er eine Long-Integer-Zahl ist, kann das Ergebnis bis zu 8 Zeichen lang sein.

• Siehe auch

OCT$, HEX$

• Beispiel

```
PRINT BIN$(8);
```

Ergebnis:

```
1000
```

BLOAD Befehl

• Syntax

BLOAD *Dateiname,Adresse*
oder
BLOAD *#Kanal,Adresse,Bytes*

• Wirkungsweise

Dieser Befehl lädt eine Binärdatei in einen zu bestimmenden Puffer oder liest eine beliebige Anzahl von Bytes von einem anzugebenden Kanal.

• Details

Der *Dateiname* ist ein String, der den Standard-AmigaDOS-Regeln entsprechen muß. Die *Adress* e muß eine Long-Integer-Zahl sein.
Die Pufferadresse muß vom Programmierer korrekt bestimmt werden; BLOAD überprüft nicht, ob es sich dabei um eine gültige Adresse im RAM handelt.
Bei der zweiten Anwendungsmöglichkeit von BLOAD muß zuvor ein Kanal mit dem OPEN-Befehl geöffnet werden. *Bytes* gibt die Anzahl der Bytes an, die gelesen werden sollen. Diese Anwendungsmöglichkeit von BLOAD ähnelt dem INPUT$-Befehl. Allerdings werden die Daten bei BLOAD in ein Array oder an eine Adresse gelesen, bei INPUT$ in einen String.

• Siehe auch

BSAVE, INPUT$, GET

• Beispiel

```
DIM A%(16000)
BLOAD "PICTURE.DMP",VARPTR(A%(0))
```

BREAK Befehl

• Syntax

```
BREAK {ON|OFF|STOP}
```

• Wirkungsweise

Bestimmt, was beim ON...BREAK-Befehl passieren soll.

• Details

Mit BREAK ON wird die Überprüfung des ON...BREAK-Befehls eingeschaltet. Dadurch kommen ON BREAK-Befehle zum Einsatz, sobald der Anweder <Ctrl>+<C> < Ⓐ >+<.> drückt.
Nach einem BREAK OFF-Befehl werden diese Tasten ignoriert, solange das Programm auf keine Eingabe wartet.
BREAK STOP bewirkt, daß sich das Programm Menüereignisse solange merkt, bis ein BREAK ON-Befehl auftaucht. ON BREAK...GOSUB wird anschließend fortgeführt. Das kann ganz nützlich sein, wenn man die ständige Abfrage nach <Ctrl>+<C> oder < Ⓐ >+<.> unterbrechen will, um zeitkritische Programmstellen abzuarbeiten.
Wenn beim Compiler die Break-Check-Compiler-Option eingeschaltet ist, hält das Programm an, wenn die <Ctrl>+<C> oder < Ⓐ >+<.> gedrückt worden sind, ohne Rücksichtnahme auf irgendwelche ON BREAK-Befehle.

• Siehe auch

ON...BREAK

• Beispiel

```
ON BREAK GOSUB Halt
BREAK ON

Halt:STOP 'hält das Programm an, wenn <Ctrl>+<C>
          oder < Ⓐ >+<.> gedrückt wurde.
```

BSAVE Befehl

• Syntax

BSAVE *Dateiname,Adresse,Länge*
oder
BSAVE *#Kanal,Adresse,Länge*

• Wirkungsweise

Dieser Befehl speichert den Inhalt eines anzugebenden Puffers auf ein Ausgabegerät, das mit *Dateiname* oder *#Kanal* definiert wird.

• Details

Es ist auch möglich – allerdings nicht sehr gebräuchlich – BSAVE auch direkt auf die parallele oder serielle Schnittstelle anzuwenden.
Sowohl die *Adresse* als auch die *Länge* des Puffers sind Long-Integer-Zahlen. Der *Dateiname* muß den AmigaDOS-Regeln entsprechen.
Alternativ dazu kann man Daten auch in eine bereits geöffnete Datei ausgeben.

• Siehe auch

BLOAD, PRINT#, WRITE.

• Beispiel

```
BSAVE "ARRAYA",VARPTR(a%(0)),
                VARPTR((UBOUND(a%)+1)*2)
```

Diese Zeile speichert das vollständige Array a%() in eine Datei namens ARRAYA.

```
DIM a%(100),b%(100)
...
OPEN "ARRAYS.DAT" FOR OUTPUT AS #1
BSAVE #1,VARPTR(a%(0)),101*2 '101 Elemente aus
                              jeweils 2 Bytes
BSAVE #1,VARPTR(b%(0)),101*2
CLOSE #1
```

Dieses Programmfragment speichert den kompletten Inhalt der Arrays a%() und b%() in die Datei ARRAYS.DAT. Achten Sie darauf, daß man nur numerische Arrays auf diese Art speichern kann.

CALL Befehl

• Syntax

```
[CALL] Unterprogrammname [(Parameter
                          [,Parameter]...)]
```

• Wirkungsweise

Ruft ein Unterprogramm, das zuvor mit SUB...END SUB definiert wurde, oder eine Bibliotheksroutine auf.

• Details

Unterprogrammname ist der Name des Unterprogramms, das aufgerufen werden soll. Achten Sie darauf, daß der Aufruf und das Unterprogramm die gleiche Anzahl an Übergabeparametern benutzen.
Sie können das Wort CALL immer dann weglassen, wenn das Unterprogramm über Parameter verfügt und wenn es sich nicht um den ersten Befehl in einer Reihe von Befehlen in einer Zeile handelt. Dies dient dazu, Konflikte mit Label-Deklarationen zu vermeiden, die ebenfalls Kennzeichner mit einem Doppelpunkt sind. Wenn CALL weggelassen wird, *müssen* auch die Klammern um die Parameter weggelassen werden.
Array-Parameter müssen in CALL-Befehlen als Array-Namen mit anschließenden Klammern () und String-Parameter als Strings übergeben werden. Die verschiedenen numerischen Parameter werden jedoch in die vom Unterprogramm geforderte Typen umgewandelt.
Parameter werden als Verweis übergeben, wenn sie nur aus einer Variablen bestehen, die den gleichen Typ besitzt wie in der Definition im Unterprogramm; außerdem dürfen VAL oder BYVAL bei der Unterprogrammdefinition nicht benutzt werden. Array-Elemente dürfen als variable Parameter übergeben werden.
Sie können eine Variable als Zahlenwert übergeben. Dazu muß sie in Klammern stehen.

• Siehe auch

SUB, FUNCTION

• *Beispiel*

```
QuickSort MeinArray(),100
CALL QuickSort(MeinArray(),100) ' Diese Aufrufe
                                  sind identisch
QuickSort(MeinArray(),100)      ' Syntax-Fehler
                                  aufgrund der
                                  Klammern
call SKIP_ONE:SKIP_ONE  'Aufruf erforderlich
                         für erstes Mal
FRED X      'Aufruf als Verweis
FRED (X)    'Aufruf als Zahlenwert
```

CALL LOC Befehl

• Syntax

```
CALL LOC Adresse [,Parameter]...
```

• Wirkungsweise

Ruft eine Maschinenspracheroutine mit der Möglichkeit der Übergabe von Parametern auf.

• Details

Dieser Befehl ist identisch zu dem Maschinensprachenbefehl CALL. Wir mußten leider die Syntax ändern, um Verwechslungen bei CALL zwischen Maschinensprache- und Unterprogrammaufrufen zu vermeiden.
Die Maschinensprachenroutine wird ab der *Adresse* ausgeführt. Folgende Parameter können auf dem Stack gefunden werden:

(a7)	long	Rücksprungadresse
4(a7)	word	Anzahl der Parameter
6(a7)	long	Zeiger auf den Parameterblock

Der Parameterblock besteht aus einer Reihe von Langwörtern, immer eins für einen übergebenen Parameter. Im Falle eines numerischen Parameters handelt es sich dabei um eine gerundete Long-Integer-Zahl. Im Falle eines String-Parameters handelt es sich um einen Zeiger auf den String, der null-terminiert sein muß.
Die Maschinenspracheroutine an der *Adresse* wird ausgeführt, und die Parameter können auf dem Stack als Long-Integers wie in C gefunden werden.
Die Maschinenspracheroutine kann mittels eines RTS-Befehls zum BASIC-Programm zurückkehren. Alle Register außer A7 und A6 können gelöscht werden.

• Beispiel

```
DIM spare%(100)
BLOAD "CALCPI",VARPTR(spare%(0))
calcpi&=VARPTR(spare%(0))
CALL LOC calcpi&,100
```

CALLS Befehl

• Syntax

`CALLS` *`Unterprogrammvariable`*

• Wirkungsweise

Ruft ein Unterprogramm indirekt über eine Variable auf, die einen Zeiger auf das Unterprogramm darstellt.

• Details

Die *Unterprogrammvariable* muß über die Funktion VARPTRS initialisiert werden, um die korrekte Einsprungadresse zu erhalten. An das Unterprogramm dürfen keine Parameter übergeben werden.
Achtung: Diese Funktion sollte nur von fortgeschrittenen Programmierern benutzt werden. Eine falsche Benutzung kann zu Systemabstürzen führen.

• Siehe auch

`VARPTRS, CALL.`

• Beispiel

```
read_char&=VARPTRS(char_from_disc)
DO
   CALLS read_char&
LOOP UNTIL no_more
.
.
SUB char_from_disc
IF EOF(2) THEN
   read_char&=VARPTRS(char_from_mem)
   CALLS read_char&
   EXIT SUB
ELSE
.
.
.
END SUB
```

CDBL Funktion

• Syntax

`CDBL(numerischer Ausdruck)`

• Wirkungsweise

Diese Funktion konvertiert einen *numerischen Ausdruck* zu einer Zahl mit doppelter Genauigkeit.

• Details

Der Effekt ist der gleiche, als wenn man einen *numerischen Ausdruck* einer Variablen mit doppelter Genauigkeit zuweist und dann diese Variable weiterbenutzt.

• Siehe auch

`MKD$, CINT, CLNG, CSNG`

• Beispiel

```
PRINT ATN(1)          'einfache Genauigkeit
PRINT CDBL(ATN(1)) 'einfache zu doppelter
                      Genauigkeit
PRINT ATN(CDBL(1)) 'doppelte Genauigkeit korrekt
```

Ergebnis:

```
 .7853982
 .7853981852531433
 .7853981633974485
```

CHAIN Befehl

• Syntax

```
CHAIN Dateiname
```

• Wirkungsweise

Mit diesem Befehl lassen sich andere Programme laden und ausführen.

• Details

Dateiname muß den AmigaDOS-Spezifikationen entsprechen. Es muß der Name eines ausführbaren (z.B. mit Doppelklick zu startenden) Programms mit all seinen Erweiterungen sein. Dabei braucht es sich nicht um ein compiliertes MaxonBASIC-Programm zu handeln.
Das Starten eines anderen Programms funktioniert nur, wenn das aufrufende Programm aus CLI bzw. Shell oder aus dem MaxonBASIC-Editor heraus gestartet wurde.
Alle Dateien, Fenster und Bildschirme werden geschlossen, bevor das Programm mit CHAIN gestartet wird.

• Siehe auch

`RUN, Execute (DOS-Library-Routine).`

• Beispiel

```
DO
   INPUT "Programm";f$
LOOP UNTIL FEXISTS(f$)
CHAIN f$
```

CHDIR Befehl

• Syntax

CHDIR *Pfadname*

• Wirkungsweise

Mit diesem Befehl wechseln Sie das aktuelle Verzeichnis.

• Details

Der *Pfadname* muß den AmigaDOS-Spezifikationen entsprechen. Pfadnamen können auch relativ (im Gegenteil zu absolut) benutzt werden. Nehmen wir z.B. an, ein Verzeichnis ":EBENE1/EBENE2" sei das aktuelle, dann würde CHDIR "/" das aktuelle Verzeichnis auf eine Ebene höher in der Hierarchie ändern (in diesem Fall ":EBENE1"). Man kann mit CHDIR z.B. auch zwei Ebenen tiefer in der Directory-Hierarchie kommen. Um von ":EBENE1" in ":EBENE1/EBENE2" zu wechseln, muß man nur CHDIR "EBENE2" angeben; CHDIR ":EBENE1/EBENE2" ist in diesem Fall nicht notwendig.

• Beispiel

```
CHDIR ":MBASIC"   'ein absoluter Pfad
CHDIR "SOURCE"    'wechselt relativ in das
                   Directory :MBASIC/SOURCE
```

CHR$ Funktion

• Syntax

`CHR$(`*`ASCII-Code`*`)`

• Wirkungsweise

Diese Funktion gibt einen einzelnen Buchstaben als String zurück, dessen *ASCII-Code* als Parameter übergeben wurde.

• Details

CHR$ wird häufig dazu benutzt, Zeichen oder Steuerzeichen, die man nicht direkt über die Tastatur erreicht, zu verwenden. Eine Anwendung wäre z.B. einen Seitenvorschub (Form Feed; ASCII-Code 12) zu einem Drucker zu schicken oder auch Sonderzeichen aus einem Zeichensatz auszugeben (z.B. ©). Bei *ASCII-Code* handelt es sich um eine Integer-Zahl.

• Siehe auch

`ASC`

• Beispiel

```
LPRINT CHR$(12); 'schickt ein Form Feed
                  zum Drucker
PRINT "Copyright ";CHR$(169);" MAXON Computer"
```

CINT Funktion

• Syntax

CINT(*numerischer Audruck*)

• Wirkungsweise

Diese Funktion konvertiert einen *numerischen Ausdruck* zu einer Integer-Zahl. Dabei werden die Nachkommastellen gerundet.

• Details

Wenn der *numerische Ausdruck* nicht im Bereich von -32768 bis 32767 liegt, wird ein Überlauf-Fehler zurückgegeben.
CINT unterscheidet sich von INT und FIX darin, daß eine gerundete Integer-Zahl erzeugt wird. Ein Beispiel, das die Unterschiede zwischen diesen drei Funktionen zeigt, finden Sie unter der INT-Funktion.
CINT entspricht der Zuweisung eines *numerischen Ausdrucks* an eine Integer-Variable.

• Siehe auch

INT, FIX, MKI$, CDBL, CSNG, CLNG

• Beispiel

```
PRINT CINT(1.5),
PRINT CINT(-1.5)
```

Ergebnis:

```
 2 -2
```

CIRCLE Befehl

• Syntax

```
CIRCLE [STEP](x_center,y_center),Radius
       [,Farbnum][,Startwinkel[,Endwinkel]
       [,Aspekt]
```

• Wirkungsweise

Zeichnet einen Kreis, eine Ellipse oder einen Kreisbogen in das aktuelle Fenster.

• Details

Wenn die optionalen Winkel-Parameter weggelassen werden, zeichnet der Befehl einen ungefüllten Kreis bzw. Ellipse in der aktuellen Vordergrundfarbe (erster Parameter beim COLOR-Befehl). Die Winkel-Parameter werden im Bogenmaß zwischen -2π und $+2\pi$ angegeben und bestimmen den Start- und Endpunkt des Kreises bzw. der Ellipse. Winkel werden entgegen dem Uhrzeigersinn beginnend auf der rechten Seite angegeben.

x_center und *y_center* bestimmen das Zentrum des Kreises. Wenn die STEP-Angabe gemacht wird, wird der Kreis/die Ellipse relativ zur aktuellen Position des Grafikpunktes gezeichnet; nach dem Zeichnen ist die Kreismitte die aktuelle Punktposition.

Wenn der *Startwinkel* oder der *Endwinkel* negativ sind, wird eine Linie von den Endpunkten zur Kreismitte gezogen, ähnlich wie man ein Stück aus einer Tortengrafik zeichnet. Die Winkel werden in diesem Falle so behandelt, als wenn sie positiv angegeben worden wären.

Mit dem *Farbnum*-Parameter wird die Farbe des Kreises/der Ellipse bestimmt. Dazu muß man zuvor mit dem PALETTE-Befehl die Farbe setzen. Wenn dieser Parameter weggelassen wird, wird die aktuelle Vordergrundfarbe benutzt.

Der *Aspekt*-Parameter dient zum Zeichnen einer Ellipse. Wenn *Aspekt* <1 ist, wird der x-Radius als *Radius* und der y-Radius als *Radius*Aspekt* genommen. Wenn *Aspekt* >=1 ist, wird der y-Radius als *Radius* und der x-Radius als *Radius/Aspekt* genommen. Der Grundwert von *Aspekt* beträgt 0.44. Dabei handelt es sich um den Wert, der auf einem Standard-Amiga mit NTSC-Monitor einen Kreis ergibt.

Gefüllte Kreise und Kreissegmente können mit dem CIRCLE-Befehl gezeichnet werden, wenn man anschließend den PAINT-Befehl benutzt.

Die *x-*, *y-*, *Radius-* and *Farbnum*-Parameter sind alle Integer-Zahlen. Die *Startwinkel-*, *Endwinkel-* und *Aspekt*-Parameter sind alle Zahlen mit einfacher Genauigkeit.

• Siehe auch

PAINT

• Examples

```
CIRCLE (200,100),75   ' zeichnet einen Kreis 75
                      ' Pixel weit, der an der
                      ' Position (200,100) seinen
                      ' Mittelpunkt hat

CIRCLE (30,30),10,,0,1.57,0.22
CIRCLE STEP(0,30),10,,0,1.57,0.22
CIRCLE STEP (30,0),10,,0,1.57,0.22
CIRCLE STEP (0,-30),10,,0,1.57,0.22
                      ' Zeichnet vier Teilkreise
```

CLEAR Befehl

• Syntax

```
CLEAR
```

• Wirkungsweise

Dieser Befehl löscht alle Variablen und schließt alle Kanäle.

• Details

Folgende Auswirkungen hat dieser Befehl:

- Alle globalen numerischen Variablen werden auf Null zurück gesetzt.
- Alle globalen String-Variablen werden zu Null-Strings ("").
- Die Inhalte aller numerischen Arrays werden auf Null zurückgesetzt.
- Die Inhalte aller String-Arrays werden zu Null-Strings.
- Alle Dateien werden geschlossen.

Dieser Befehl wurde nur aus Kompatibilitätsgründen zu älteren BASIC-Dialekten in MaxonBASIC aufgenommen. Normalerweise sollte man lieber selbst kontrolliert Variablen löschen und Kanäle schließen.

• Siehe auch

```
RESET.
```

• Beispiel

```
IF a$="ZERO" THEN CLEAR
```

CLNG Funktion

• Syntax

CLNG (*numerischer Ausdruck*)

• Wirkungsweise

Diese Funktion wandelt einen *numerischen Ausdruck* in einen Long-Integer-Wert um; die Nachkommastellen werden gerundet.

• Details

Wenn der *numerische Ausdruck* nicht im Bereich von –2147483648 bis 2147483647 liegt, tritt ein Überlauf-Fehler (Overflow) auf.
CLNG ähnelt der CINT-Funktion, gibt aber im Gegensatz zu CINT einen Long-Integer-Wert zurück. CLNG kann häufig dazu benutzt werden, sicherzustellen, daß kein Überlauf-Fehler auftritt, wenn man das Produkt einiger Integer-Zahlen einer Long-Integer-Zahl zuweist.
CLNG entspricht der Zuweisung eines *numerischen Ausdrucks* an eine Long-Integer-Variable.

• Siehe auch

INT, FIX, MKI$, CDBL, CSNG, CINT.

• Beispiel

```
x&=CLNG(i%)*1000

'ist normalerweise besser als

x&=i%*1000

'da letzteres einen Überlauf-Fehler ergibt,
'wenn i% = 33 oder mehr.
```

CLOSE Befehl

• Syntax

```
CLOSE [[#]Kanalnummer[,[#]Kanalnummer]...]
CLOSE {ON|OFF|STOP}
```

• Wirkungsweise

Dieser Befehl beendet eine Ein-/Ausgabe auf eine bestimmte Datei oder ein Gerät oder kontrolliert alternativ auch Ereignisse von Fenstern, die vom Anwender geschlossen werden.

• Details

Bei der ersten Variante dieses Befehls handelt es sich um das Gegenteil des OPEN-Befehls. Die *Kanalnummer* ist die Nummer, die zuvor bereits im OPEN-Befehl für eine bestimmte Datei oder ein Gerät angegeben wurde.
CLOSE ohne einen Parameter schließt alle Dateien und Gerätekanäle, die zuvor mit OPEN geöffnet wurden.
Wurde eine Datei erst einmal mit CLOSE geschlossen, kann die Kanalnummer direkt wieder benutzt werden, um eine andere Datei oder einen anderen Gerätekanal zu öffnen.
Wenn man eine Datei oder ein Gerät schließt, das zuvor zum Schreiben von sequentiellen Daten geöffnet wurde, wird zuerst der Puffer vor dem eigentlichen Schließen gesichert.
Wenn man eine Datei schließt, die zuvor für einen wahlfreien Zugriff (Random Access) geöffnet wurde, werden automatisch alle Variablen, die mit dem FIELD-Befehl mit dieser Datei verbunden wurden, gelöscht.
CLEAR, END/SYSTEM, RESET, STOP und RUN schließen alle Dateien und Gerätekanäle automatisch.
Die zweite Variante dieses Befehls überwacht das Verhalten eines Programms, wenn der Anwender ein Fenster schließt. Normalerweise (entspricht CLOSE ON und ist die Grundeinstellung) schließt der Anwender ein Fenster, und das Programm erhält darüber keine Nachricht. CLOSE OFF bewirkt nun, daß ein solches Schließen (window close event) ignoriert wird.
Bei CLOSE STOP wird das Schließen (window close event) solange ignoriert, bis ein CLOSE ON-Befehl ausgeführt wurde; weil z.B. eine zeitkritische Programmstelle ausgeführt wird und alle Rechenzeit benötigt wird.

• Siehe auch

OPEN, ON...CLOSE

• Beispiel

```
CLOSE #1,#2
```

CLS Befehl

• Syntax

```
CLS
```

• Wirkungsweise

Löscht das aktuelle Ausgabefenster und setzt den Cursor in die linke obere Ecke.

• Details

Es wird nur das aktuelle Fenster gelöscht, alle anderen bleiben davon unberührt.

COLLISION Befehl

• Syntax

```
COLLISION {ON|OFF|STOP}
```

• Wirkungsweise

Verändert das Verhalten bei Kollisionen in Zusammenhang mit dem ON...COLLISION-Befehl (collision event trap).

• Details

Mit COLLISION ON wird die Kollisionskontrolle (collision event checking) eingeschaltet. Das bewirkt, daß ON...COLLISION-Befehle ausgeführt werden, sobald ein Objekt mit dem Rand oder einem anderen Objekt kollidiert.
Mit COLLISION OFF wird die Kollisionskontrolle ausgeschaltet. Kollisionen werden jetzt ignoriert.
COLLISION STOP bewirkt, daß Menü-Ereignisse (menu events) zwischengespeichert werden, bis ein COLLISION ON-Befehl ausgeführt wird. Der ON COLLISION...GOSUB-Befehl wird dann fortgesetzt. COLLISION STOP ist notwendig, wenn z.B. eine zeitkritische Programmstelle ausgeführt wird und alle Rechenzeit dafür benötigt wird.

• Siehe auch

```
ON...COLLISION, OBJECT, COLLISION-Funktion
```

• Beispiel

```
COLLISION ON
COLLISION OFF
COLLISION STOP
```

COLLISION Funktion

• Syntax

`COLLISION(Objekt_ID)`

• Wirkungsweise

Liefert Informationen über Objektkollisionen, die aufgetreten sind.

• Details

Es gibt drei verschiedene Möglichkeiten, die COLLISION-Funktion aufzurufen:

COLLISION(-1) gibt die Fenster-ID zurück, in dem die nächste Kollision passiert ist.
COLLISION(0) gibt die ID von dem Objekt zurück, das in die nächste Kollision verwickelt ist.
COLLISION(n) dient dazu, herauszufinden, in welche Art von Kollision das Objekt mit der ID n verwickelt ist (wobei n>0 sein muß); dadurch wird die Kollision auch von der Liste der noch nicht abgeschlossenen Ereignisse (pending events) entfernt. n erhält man normalerweise über den Befehl COLLISION(0). Mittels COLLISION(n) bekommt man einen der folgenden Werte zurück:

–1	oberer Rand
–2	linker Rand
–3	unterer Rand
–4	oberer Rand
>0	Objekt-ID des Objektes, mit dem Objekt n kollidiert ist

MaxonBASIC verwaltet mit seinem Ereignispuffer bis zu 16 Ereignisse, genauso wie AmigaBASIC.

• Siehe auch

ON...COLLISION, OBJECT, COLLISION-Befehl.

• Beispiel

```
obj%=COLLISION(0)
IF obj%>0 THEN
   sort%=COLLISION(obj%)
   SELECT CASE sort%
   CASE -1: 'Kollision mit oberem Rand
   CASE -2: 'Kollision mit linkem Rand
   CASE -3: 'Kollision mit unterem Rand
   CASE -4: 'Kollision mit rechtem Rand
   CASE ELSE  'Kollision mit einem anderem Objekt
   END SELECT
END IF
```

COLOR Befehl

• Syntax

COLOR
[*Vordergrundfarbe*] [Hintergrundfarbe] [,*Zeichenmodus*]

• Wirkungsweise

Setzt die Farben, die in dem aktuellen Fenster benutzt werden sollen.

• Details

Hiermit wird die *Vordergrundfarbe* für das aktuelle Fenster bestimmt; sie wird für Punkte, Linien, Texte und Füllungen benutzt. Die *Hintergrundfarbe* gibt die Farbe des Bildschirmhintergrunds an.
Die richtige Farbnummer kann mit dem PALETTE-Befehl und dem Palette Preferences Tool ausgewählt werden.
Die voreingestellten Farbwerte des Workbench-Bildschirms unter Workbench 2 und 3 sind:

0 Grau
1 Schwarz
2 Weiß
3 Blau

Es kann allerdings vorkommen, daß diese Farben vom Anwender anders zusammengestellt wurden; das ist Geschmackssache. In diesem Fall stimmen die Farbangaben natürlich nicht mehr. Wenn Sie ganz bestimmte Farben benötigen, sollten Sie einen eigenen Bildschirm öffnen, um dies zu erreichen. Die Farben unter der Workbench 1.3 stimmen mit den angegebenen Werten nicht überein.
Wenn ein Farbparameter weggelassen wird, bleibt der Farbwert an dieser Stelle derselbe, wie er aktuell bestimmt ist.
Bei dem optionalen *Zeichenmodus* handelt es sich um einen speziellen MaxonBASIC-Parameter, mit dem sich der Zeichenmodus setzen läßt, ohne die Graphics-Library direkt aufzurufen. Folgende Zeichenmodi sind möglich:

0 JAM1
Die Einsen des aktuellen Bitmusters werden mit der Vordergrundfarbe gezeichnet, die Nullen werden ignoriert.
1 JAM2
Die Einsen des aktuellen Bitmusters werden mit der Vordergrundfarbe gezeichnet, die Nullen mit der Hintergrundfarbe. Das ist die Grundeinstellung.

2 KOMPLEMENT
Für jedes einzelne Bit, das gezeichnet wird, wird die Farbe komplementiert; z.B. wenn man 4 Farben benutzt: Farbe 0 wird zu Farbe 3, Farbe 1 wird zu Farbe 2 und umgekehrt.
4 INVERSEVIDEO und JAM1
Die Pixel, die den Text umgeben, werden in der Vordergrundfarbe gezeichnet und die aktuellen Textpixel bleiben so, wie sie sind.
5 INVERSEVIDEO und JAM2
Der Text wird in der Hintergrundfarbe geschrieben und von der Vordergrundfarbe umgeben. Einfacher gesagt: der Bildschirm wird invertiert.

• *Beispiel*

```
COLOR 2,3
CLS
PRINT "Füllt das Fenster mit schwarzem Hinter-
grund, die Vordergrundfarbe ist Orange"
COLOR ,,5
PRINT "Das ist invertiert"
COLOR ,,2
PRINT "Wieder normal"
```

COMMAND$ Funktion

• Syntax

COMMAND$

• Wirkungsweise

Diese Funktion gibt die Kommandozeile des Programms zurück.

• Details

Diese Funktion ermöglicht einem Programm, seine Kommandozeile zu bearbeiten, wenn es vom CLI aus gestartet wurde.
Alle Leerzeichen, die vor einem Befehl in der Kommandozeile stehen, werden gelöscht.
Wenn ein Programm von der Workbench aus läuft, gibt die Funktion die Namen jeder Datei, die ausgewählt wurde, zurück. Ab Workbench 2 erfolgt die Angabe des vollen Pfadnamens.
Unter Workbench 1 wird nur der letzte Teil des Pfades, also nur der Dateiname selbst, zurückgegeben. Folglich läßt sich die Funktion in diesem Falle nur dann sinnvoll einsetzen, wenn die ausgewählte Datei im selben Unterverzeichnis steht.

• Beispiel

```
OPEN COMMAND$ AS #1 ' ein Dateiname wird
                    ' übergeben, um eine Datei
                    ' zu öffnen
```

COMMON SHARED Befehl

• Syntax

```
COMMON SHARED Variable [(Indizes…)][,Variable
             [(Indizes…)]]…
```

• Wirkungsweise

Dieser Befehl bewirkt, daß globale Variablen für jedes Unterprogramm gültig sind und dort benutzt werden können.

• Details

Dieser Befehl dient in erster Linie der Kompatibilität zu QuickBASIC und entspricht im wesentlichen dem DIM SHARED-Befehl, mit Ausnahme, daß die Indizes der Arrays ignoriert werden. Will man Arrays in diesem Befehl benutzen, müssen diese, bevor man den COMMON SHARED-Befehl anwendet, mit dem DIM-Befehl dimensioniert werden.

• Beispiel

```
DIM glob(20)
COMMAND SHARED debug_flag,glob(1)
```

entspricht

```
DIM SHARED debug_flag,glob(20)
```

Sowohl *debug_flag* als auch *glob()* können nun im Programm benutzt werden.

CONST Befehl

• Syntax

```
CONST Name=Integer_Konstante
            [,Name=Integer_Konstante]...
```

• Wirkungsweise

Mit CONST definiert man symbolische Konstanten, die dann statt Integer-Variablen genutzt werden können.

• Details

Konstanten müssen vor ihrer Anwendung zunächst einmal deklariert werden. Ihr *Name* muß eine gültige Integer- oder Long-Integer-Variable sein. *Integer_Konstante* muß eine Integer-Zahl (oder Long-Integer) oder eine Buchstabenkonstante mit optional vorangestelltem Minuszeichen sein.
Konstanten dienen dazu, Lesezugriffe auf Datenstrukturen im Speicher zu erleichtern.
Konstanten dürfen in Unterprogrammen und vom Anwender definierten Funktionen benutzt werden, ohne sie zuvor als SHARED zu deklarieren. Diese Methode entspricht *nicht* der der letzten Version des Microsoft BASIC; dort sind Konstanten in Unterprogramm lokal.

• Siehe auch

DEFINT

• Beispiel

```
DEFINT a-z
CONST langconst&=&h123456
CONST weiss=0,schwarz=1,rot=2,gruen=3
COLOR rot,gruen,schwarz
CONST beginn="A"%,schluss="Z"%
IF (c%>=beginn) AND (c%<=schluss) THEN
   PRINT "Es ist ein Kleinbuchstabe"
```

COS Funktion

• Syntax

`COS(numerischer_Ausdruck)`

• Wirkungsweise

Diese Funktion gibt den Cosinus eines *numerischen Ausdrucks* zurück, der im Bogenmaß angegeben werden muß.

• Details

Normalerweise arbeitet COS mit einfacher Genauigkeit. Wenn der *numerische Ausdruck* allerdings mit doppelter Genauigkeit angegeben wird, arbeitet COS ebenfalls doppelt genau.

• Siehe auch

`SIN, TAN, ATN`

• Beispiel

```
PRINT COS(0)
```

Ergebnis:

```
1
```

CSNG Funktion

• Syntax

`CSNG(numerischer_Ausdruck)`

• Wirkungsweise

Diese Funktion wandelt einen *numerischen Ausdruck* in eine Zahl mit einfacher Genauigkeit um.

• Details

CSNG hat denselben Effekt, wie wenn man einen *numerischen Ausdruck* einer Variablen mit einfacher Genauigkeit zuweist und die Variable anschließend benutzt.

• Siehe auch

`MKS$, CDBL, CINT, CLNG`

• Beispiel

```
PRINT SIN(1#)
PRINT CSNG(SIN(1#))
```

Ergebnis:

```
.8414709848078965
.841471
```

CSRLIN Funktion

• Syntax

```
CSRLIN
```

• Wirkungsweise

Diese Funktion gibt die Zeile der aktuellen Cursor-Position zurück.

• Details

Um die aktuelle Spaltenposition zu bekommen, muß man die POS-Funktion verwenden. Bei dem Rückgabewert handelt es sich um eine Integer-Zahl. Befindet sich der Cursor am Anfang des Bildschirms (oder Fensters), wird eine 1 zurückgegeben, entsprechend dem LOCATE-Befehl. Wurde das aktuelle Ausgabefenster geschlossen, wird ebenfalls eine 1 zurückgegeben.

• Siehe auch

```
POS, PRINT, LOCATE
```

• Beispiel

```
L%=CSRLIN
C%=POS(0)
PRINT "Der Cursor befindet sich in
       Zeile";L%;"und Spalte";C%
```

Ergebnis (wenn sich der Cursor in der Zeile 2 und Spalte 3 befindet):

```
Der Cursor befindet sich in Zeile 2 und Spalte 3
```

CURDIR$ Funktion

• Syntax

```
CURDIR$
```

• Wirkungsweise

Die Funktion gibt den Namen des aktuellen Verzeichnisses (Directory) zurück.

• Details

Der String, der zurückgegeben wird, beinhaltet auch immer den Laufwerksnamen. Am Ende des Strings steht *kein* Schrägstrich (/). Dieser Befehl funktioniert erst ab Workbench 2.0; bei früheren Workbench-Versionen wird ein Null-String zurückgegeben

• Siehe auch

```
CHDIR.
```

• Beispiel

```
' wenn MAXON: das aktuelle Laufwerk
' und MBasic das aktuelle Verzeichnis ist
PRINT CURDIR$
```

Ergebnis:

```
MAXON:MBasic
```

CVD,CVI,CVL,CVS Funktionen

• Syntax

```
CVD(8-Byte-String einer Fließkomazahl
    mit doppelter Genauigkeit)
CVI(2-Byte-String einer Integer-Zahl)
CVL(4-Byte-String einer Long-Integer-Zahl)
CVS(4-Byte-String einer Fließkomazahl
    mit einfacher Genauigkeit)
```

• Wirkungsweise

Diese Funktionen geben die internen numerischen Werte von Byte-Strings zurück.

• Details

Diese Funktionen stellen das Gegenstück zu MKD$, MKI$, MKL$ und MKS$ dar. Sie verändern nicht den Wert der aktuellen Daten, sie teilen MaxonBASIC lediglich mit, daß sie anders zu interpretieren sind.
Das häufigste Anwendungsgebiet für diese Funktionen liegt bei der Programmierung von Dateien mit wahlfreiem Zugriff (Random Access). Dabei werden die Daten mit den MKD$-, MKI$-, MKL$- oder MKS$-Funktionen als String gespeichert und können mit den CV-Funktionen zurückgewandelt werden.

• Beispiel

```
D#=9/11
I%=10
L&=42
S!=5/7
PRINT D#,I%,L&,S!
OPEN "TEST.DAT" FOR OUTPUT AS #1
PRINT #1,MKD$(D#);MKI%(I%);MKL(L&);MKS(S!)
CLOSE #1
'Die konvertierten Daten sind jetzt binär. Bei
'der Ausgabe auf dem Bildschirm würden diese
'Werte als seltsame Steuerzeichen angezeigt.
OPEN "TEST.DAT" FOR INPUT AS #1
D1$=INPUT$(8,#1)
I1$=INPUT$(2,#1)
L1$=INPUT$(4,#1)
S1$=INPUT$(4,#1)
CLOSE #1
PRINT CVD(D1$),CVI(I1$),CVL(L1$),CVS(S1$)
```

Ergebnis:

```
 .8181818127632141    10 42 7142857
 .8181818127632141    10 42 7142857
```

DATA Befehl

• Syntax

`DATA Konstante [,naechste_Konstante]...`

• Wirkungsweise

Mit diesem Befehl lassen sich Daten definieren, die dann mit dem READ-Befehl gelesen werden können.

• Details

Der DATA-Befehl führt keine Programme aus. Er kann soviele Konstanten enthalten, wie in eine Zeile passen. Der READ-Befehl liest DATA-Konstanten nacheinander in ihrer Reihenfolge ein. Folgen mehrere DATA-Befehle aufeinander, werden sie als ein zusammenhängender String von Konstanten angesehen.
Konstanten können jedem Variablentyp entsprechen. Soweit es notwendig ist, werden die Daten, die vom READ-Befehl gelesen werden sollen, in das richtige Datenformat gewandelt.
Es ist nicht nötig, String-Konstanten in DATA-Befehlen in Anführungszeichen zu setzen, es sei denn, der String enthält Doppelpunkte, Kommata, Leerzeichen am Anfang oder Ende, oder er könnte unerwünschterweise als numerischer Wert gedeutet werden.
MaxonBASIC verwaltet Daten intern so, wie es ihm am geeignetsten erscheint; so wird 1.12345 z.B. als eine Zahl mit einfacher Genauigkeit verwaltet, und benötigt somit nur 4 Bytes an Speicherplatz. Will man diesen Wert dann in eine Variable mit einfacher Genauigkeit einlesen, muß man ihn – im Gegensatz zu vielen BASIC-Dialekten – nicht erst von ASCII- in Binärcode umwandeln. Will man eine Zahl in eine String-Variable einlesen, wird sie automatisch zu einem String zurückgewandelt.
Dies hat den Nachteil, daß die String-Darstellung nicht der Originaldarstellung entspricht, z.B. wird aus 11E1 dann 110. Um dies zu vermeiden, müssen solche Strings in Anführungszeichen gesetzt werden.
Ein Leerelement (,, = aufeinanderfolgende Kommata) in der Konstantenliste ist erlaubt. Wenn ein solches Leerelement auftaucht, wird im Falle einer numerischen Variablen der Wert 0 zugewiesen, eine String-Variablen erhält einen Null-Wert.
Normalerweise lassen sich DATA-Befehle mit READ nur einmal einlesen. Will man sie trotzdem mehrmals lesen, muß zuvor der RESTORE-Befehl ausgeführt werden.
DATA-Befehle dürfen nicht in vortokenisierten Dateien vorkommen.

• Siehe auch

READ, RESTORE, OPEN, INPUT$

• Beispiel

```
DATA 2.2,1.6,"Hallo"
READ a,b%,c$
PRINT a;b%;c$
```

Ergebnis:

```
 2.2  2 Hallo
```

Das DATA-Element „1.6“ wurde gerundet, bevor es der Integer-Variablen b% zugewiesen wurde, und hat somit den Wert „2“.

DATE$ Funktion

• Syntax

```
DATE$
```

• Wirkungsweise

Diese Funktion gibt das aktuelle Datum zurück, sofern Ihr Amiga über eine akkugepufferte Echtzeituhr verfügt.

• Details

Der Rückgabewert ist ein aus zehn Zeichen bestehender String. Das Format sieht folgendermaßen aus: *mm-tt-jjjj*.

• Siehe auch

`TIME$`.

• Beispiel

```
PRINT "Heute ist folgendes Datum: ";DATE$
```

Ergebnis (sofern heute der 24. Februar 1993 ist):

```
Heute ist folgendes Datum: 02-24-1993
```

DECLARE statement

• Syntax

```
DECLARE [SUB|FUNCTION]Routinenname
        [(Parameterliste)]
```

oder

```
DECLARE [SUB|FUNCTION]Routinenname
        [(Parameterliste)] LIBRARY
```

oder

```
DECLARE [SUB|FUNCTION]Routinenname [CDECL][ALIAS
        "externer_Name"][(Parameterliste)]
```

• Wirkungsweise

Meldet ein BASIC-Unterprogramm oder eine Funktion an, bevor sie definiert wurde, zeigt, daß eine Bibliotheksfunktion als Unterprogramm fungiert oder meldet ein extern hinzuzulinkendes Unterprogramm oder eine entsprechende Funktion an.

• Details

Die erste Anwendungsweise meldet ein BASIC-Unterprogramm oder eine Funktion an, so daß man sie bereits benutzen kann, bevor sie definiert wurden. *Routinenname* und *Parameterliste* müssen genauso angegeben werden, wie sie zuvor in dem SUB- oder FUNCTION-Befehl definiert wurden. Die Anwendung von DECLARE für Unterprogramme ist in MaxonBASIC optional, in der letzten Version von Microsoft BASIC allerdings erforderlich. Bei allen BASIC-Dialekten müssen FUNCTION-Befehle definiert werden, bevor sie angewandt werden. Dies dient dazu, daß sie nicht mit normalen Variablen oder Arrays verwechselt werden.

Wenn man ein Unterprogramm oder eine Funktion mit DECLARE anmeldet, sie aber nicht definiert, gibt der Compiler einen Fehler Nr. 48 aus, es sei denn, man hat die Option *Allow undefined sub-programs* (UNDEFSUBS) aktiviert.

Wenn man den LIBRARY-Parameter benutzt wird die *Parameterliste* ignoriert.

Wenn CDECL angegeben wird, heißt das, daß eine externe Routine in C oder Assembler zu dem Programm hinzugelinkt wird. Die optionale *Parameterliste* hat genau das gleiche Format wie BASIC-Unterprogramme oder Funktionen. MaxonBASIC erzeugt bei Verwendung dieses Befehls eine linkfähige Datei im gleichen Amiga-Format, als wenn sie ausführbar wäre.

Wird der ALIAS-Parameter verwandt, wird der Name der externe Routine der *externe_Name*. Dabei wird auf Groß-/Kleinschrift geachtet. Wenn der ALIAS-Parameter keinen Namen für die Routine im BASIC findet und keine Typenspezifikation angegeben wurde, wird automatisch Kleinschrift verwendet und ein Unterstrich (_) an den Anfang des Namens gestellt. Dies dient dem einfacheren Hinzulinken von Lattice/ SAS-C-Routinen.

• Siehe auch

FUNCTION, SUB

• Beispiel

```
LIBRARY "graphics.library"
DECLARE FUNCTION SetSoftStyle& LIBRARY

DECLARE FUNCTION doppelt(BYVAL i)

PRINT doppelt(23)

FUNCTION doppelt(BYVAL i)
doppelt=i*2
END FUNCTION

DECLARE SUB Extern CDECL (BYVAL i%)

Extern 23
' hiermit wird _extern mit einem Parameter von
' 23 auf dem Stack aufgerufen

DECLARE FUNCTION Comp CDECL(BYVAL a!,BYVAL b!)
DECLARE SUB assem CDECL ALIAS "assembler" _
        (BYVAL parameter1&,parameter2)
DECLARE SUB keine_parameter CDECL ALIAS "ASSEM"
DECLARE FUNCTION cfunktion& CDECL
```

DECR Befehl

• Syntax

`DECR numerische_Variable`

• Wirkungsweise

Zieht Eins von einer bestimmten Variablen ab.

• Details

numerische_Variable kann eine einfache numerische Variable oder ein numerisches Array-Element sein. Wenn man eine einfache Variable X dekrementieren will, ist es stilistisch sicherlich besser X=X–1 als DECR X zu benutzen. Hat man jedoch einen komplexen Array-Ausdruck wie in unserem Beispiel unten, sollte man lieber DECR nehmen.

• Siehe auch

`INCR`

• Beispiel

```
DECR A(B(I,j+5))
'entspricht A(B(I,j+5))=A(B(I,j+5))-1
```

DEF FN Befehl

• Syntax

```
DEF FNFunktionsname[(Parameterliste)]=Ausdruck
```
oder
```
DEF FNFunktionsname[(Parameterliste)]
        [LOCAL Variablenliste]
        [STATIC Variablenliste]
        [SHARED Variablenliste]
.
.
Befehle
.
.
[EXIT DEF]
[FNFunktionsname=Ausdruck]
.
.
END DEF
```

• Wirkungsweise

Definiert ein- oder mehrzeilige Funktionen mit Parametern,

• Details

Normalerweise empfehlen wir lieber FUNCTION anstatt DEF FN zu verwenden, da mit FUNCTION definierte Funktionen leichter in ein Unterprogramm verwandelt werden können und umgekehrt.
Der *Funktionsname* gibt den Namen der Funktion an und darf nur einmal in dem Programm vorkommen. Die optionale *Parameterliste* darf eine beliebige Anzahl von durch Kommata getrennten Variablennamen enthalten. Jeder Variablen kann ein VARPTR vorangestellt werden, um den Variablenwert zu erhalten.
Der *Funktionsname* muß ein Variablenname sein. Der Variablentyp, der zurückgegeben wird, entspricht den allgemeinen Regeln für Variablen; so gibt FNfred% eine Integer-Zahl zurück. FNjohn wird normalerweise eine Fließkommazahl mit einfacher Genauigkeit zurückgeben; es sei denn, man hat zuvor den Buchstaben J mit dem DEFtype-Befehl definiert. Man muß darauf achten, daß DEF FN-Funktionsnamen immer mit FN beginnen müssen.

In Funktionsdefinitionen sind Variablen immer als global definiert, es sei denn man gebraucht den STATIC- oder LOCAL-Befehl, um dies zu verhindern.
Funktionsdefinitionen dürfen rekursiv sein und Funktionen dürfen aufgerufen werden, bevor sie deklariert wurden. Allerdings können Funktionsdefinitionen nicht in anderen Funktionen oder Unterprogrammen gemacht werden.
FN*Funktionsname=Ausdruck* wird dazu benutzt, Werte von mehrzeiligen Funktionen zurückzubekommen. Man darf diesen Befehl mehrmals in einer Funktionsdefinition benutzen.
EXIT DEF bewirkt, daß die aktuelle Funktion sofort verlassen wird.
Wenn man mehrzeilige Funktionen in einem Ausdruck verwendet, muß man darauf achten, keine unerwünschten Nebeneffekte zu bekommen. Z.B.:

```
DEF FNx
  j=j+1
  FNx=j
END DEF
j=4
PRINT FNx+j
```

liefert nicht das gleiche Resultat wie

```
j=4
PRINT j+FNx
```

da die Variable j bereits verändert wurde, als die Funktion aufgerufen wurde.
Genauso ist es gefährlich, Ein-/Ausgabebefehle in mehrzeiligen Funktionen zu verwenden, wenn man sie aus einem anderen Ein-/Ausgabebefehle aus aufruft. Z.B.:

```
PRINT 54,FNquatsch,j
DEF FNquatsch
PRINT " Die Funktion quatsch wurde aufgerufen."
FNquatsch=1.4
END DEF
```

ist nicht sehr sinnvoll.
Das Beste ist niemals Funktionen in Ein-/Ausgabebefehlen aufzurufen *oder* niemals etwas in Funktionen ein- oder auszugeben.

• Siehe auch

FUNCTION

• Beispiel

```
DEF FNahex$(a&)="&H"+hex$(a&)
DEF FNlanghex$(a&)
STATIC a$
a$=hex$(a&)
IF LEN(a$)<8 THEN
  a$=STRING$(8-LEN(a$),"0")+a$
END IF
FNlanghex$=a$
END DEF
PRINT FNahex$(42),FNlanghex$(42)
```

DEFDBL, DEFINT, DEFLNG, DEFSNG, DEFSTR Befehle

• Syntax

```
DEFDBL Buchstaben [,Buchstaben]...
DEFINT Buchstaben [,Buchstaben]...
DEFLNG Buchstaben [,Buchstaben]...
DEFSNG Buchstaben [,Buchstaben]...
DEFSTR Buchstaben [,Buchstaben]...
```

• Wirkungsweise

Diese Befehle definieren, ob Variablen und Funktionen eine Integer- oder Long-Integer-Zahl, eine Zahl mit einfacher oder doppelter Genauigkeit oder ein String sind.

• Details

Das Format der *Buchstaben* lautet *Buchstaben1* [*-Buchstaben2*]. Groß-/Kleinschreibung ist erlaubt. Jeder Variablen- oder Funktionsname, der mit einem Buchstaben beginnt, der in einem DEFtype-Befehl definiert wurde, **bestimmt auch gleich den Typ der Variablen**: DBL für eine Zahl mit doppelter Genauigkeit, INT für eine Integer-Zahl, LNG für eine Long- Integer-Zahl, SNG für eine Zahl mit einfacher Genauigkeit, oder STR für einen String.
Die Typendeklarationszeichen #,%,&,!,$ haben Vorrang vor jedem DEFtype-Befehl.
Achtung: Die Variablen I#, I%, I&, I!, und I$ sind alles verschiedene Variablen und haben mit hoher Wahrscheinlichkeit auch verschiedene Werte.

• Beispiel

```
DEFINT a-z ' Sollte man gleich am Anfang seines
           ' Programms definieren, damit man nur
           ' dann Fließkommazahlen verwendet, wo
           ' sie benötigt werden. So vermeidet
           ' man viele %-Zeichen in den Namen.
DEFINT i-n
DEFSNG a-h,o-z ' für alte FORTRAN-Programmierer
```

DIM Befehl

• Syntax

```
DIM [SHARED]Variable [(Indizes...)]
    [,Variable [(Indizes...)]]...
```

• Wirkungsweise

Dieser Befehl bestimmt die maximalen Anzahl der Indizes von Array-Variablen und reserviert den benötigten Speicher für sie.

• Details

Die maximale Anzahl an Indizes, die einem Array ohne DIM-Befehl zur Verfügung stehen, ist 10. Die maximale Anzahl an Dimensionen, die ein Array haben kann, beträgt 31. (Das wäre ein Minimum von 4 Gigabyte Speicher, wenn jeder Index über mehr als ein Element verfügt.) Indexwerte können Integer- oder Long-Integer-Ausdrücke sein.
Es spricht für guten Programmierstil, wenn man auch bei weniger als 10 Indizes den DIM-Befehl benutzt. Man kann dann die Array-Prüffunktion (NOARRAY) ausschalten. Um sicherzugehen, daß man nicht ein Array aus Versehen dimensioniert, sollte man die automatische DIM-Array-Option (NOAUTODIM) ausschalten.
Den kleinsten Wert für jeden Index kann man über den OPTION BASE-Befehl bestimmen.
Wenn man REM $DYNAMIC in seinem Programm benutzt, werden alle danach dimensionierten Arrays dynamisch und können mit ERASE, REDIM und REDIM PRESERVE verändert werden. Standardmäßig bzw. nach REM $STATIC sind die Arrays statisch, wenn bei der Dimensionierung Konstanten verwendet werden. Statische Arrays können nicht mit ERASE gelöscht oder mit REDIM neu dimensioniert werden; sie sind allerdings deutlich schneller.
Nach der Definition haben die Elemente eines numerischen Arrays alle den Wert 0, String-Arrays sind leer.
DIM SHARED dient dazu, Arrays und Variablen zu deklarieren, auf die aus Unterprogrammen heraus zugegriffen werden kann, ohne daß man in jedem Unterprogramm einen SHARED-Befehl benutzen muß.

• Siehe auch

ERASE, REDIM, STATIC, LOCAL

• Beispiel

```
DIM R(14)
FOR I%=1 TO 14
   R(I%)=I%
   PRINT R(I%);
NEXT I%
```

Ergebnis:

```
1  2  3  4  5  6  7  8  9  10  11  12  13  14
```

```
DIM SHARED debug_flag 'auf debug_flag kann jetzt
                      'von überall im Programm
                      'zugegriffen werden
CONST max=100
var=100
DIM a(100),b(max),c(var),d(max*2)
' a & b sind statische Arrays, c & d dynamische
ERASE c
REDIM d(var)
'ist möglich
ERASE a
REDIM b(var)
'ist nicht möglich
```

DO...LOOP Befehl

• Syntax

```
DO [{WHILE|UNTIL} boolscher_Ausdruck]
.
.
[EXIT {DO|LOOP}]
.
.
{WEND|LOOP}[{WHILE|UNTIL} boolscher_Ausdruck]
```

• Wirkungsweise

Wiederholt die Befehle in der DO...LOOP-Schleife solange die Bedingungen TRUE (WHILE) oder FALSE (UNTIL) sind.

• Details

Der *boolsche_Ausdruck* muß ein numerischer Ausdruck sein. Hat er den Wert 0 ist er FALSE, alle anderen Werte ergeben TRUE.
Wenn ein EXIT DO- oder EXIT LOOP-Befehl in der Schleife enthalten ist, wird die Bedingung am Ende der Schleife überprüft. EXIT DO ist die bevorzugte Form und wird auch in Microsoft BASIC benutzt.
DO...LOOPs bilden in der Regel die innersten Schleifen in einer Schleifenkonstruktion. Wenn man eine äußere Schleife beenden will, ist es besser, eine REPEAT-Schleife zu verwenden, allerdings ist diese im Gegensatz zu DO...LOOP nicht kompatibel zu Microsoft BASIC.
An beiden Enden der Schleife darf – sofern gewünscht – eine Bedingung stehen, muß aber nicht. Dies macht sie wesentlich flexibler als eine WHILE...WEND-Schleife.

• Siehe auch

WHILE, REPEAT.

• Beispiele

```
i=0
DO
   i=i+1
   PRINT i;
LOOP UNTIL i>=10
```

Ergebnis:

```
 1   2   3   4   5   6   7   8   9   10
```

```
i=0
DO
   i=i+1
   IF i>10 THEN EXIT DO
   PRINT i
LOOP
```

Ergebnis:

```
 1   2   3   4   5   6   7   8   9   10
```

END Befehl

• Syntax

```
END
END DEF
END FUNCTION
END IF
END REPEAT Name
END SELECT
END SUB
```

• Wirkungsweise

Beendet ein BASIC-Programm, eine Funktionsdefinition, einen IF...THEN...ELSE-Block, einen SELECT-Befehl, eine REPEAT-Schleife oder ein Unterprogramm.

• Details

Der END DEF-Befehl gibt das Ende von DEF FN an.
END IF beendet einen IF...THEN...ELSE-Block.
END FUNCTION gibt das Ende einer FUNCTION-Definition an.
END REPEAT *Name* beendet eine REPEAT-Schleife.
END SELECT beendet einen SELECT CASE-Befehl.
END SUB beendet ein BASIC-Unterprogramm; es bildet das Gegenteil zu SUB. Näheres findet man unter diesem Befehl.
END ohne Zusatz beendet das ganze BASIC-Programm. Der END-Befehl kann irgendwo im Quelltext stehen.

• Siehe auch

EXIT

• Beispiel

```
FUNCTION faktor&(BYVAL N%)
   IF N% <= 1 THEN
        faktor&=1
   ELSE
        faktor& = faktor&(N%-1)*N%
   END IF
END FUNCTION
```

EOF Funktion

• Syntax

`EOF(Kanalnummer)`

• Wirkungsweise

Diese Funktion überprüft, ob das Ende eines Datenkanals erreicht wurde (End-Of-File).

• Details

Für TRUE wird eine -1 zurückgegeben werden; für FALSE eine 0. Eine übliche Anwendung von EOF ist das Ende einer Datei abzufragen, wenn Daten in eine sequentielle Datei geschrieben werden. Wird in eine Datei mit wahlfreiem Zugriff (Random Access) geschrieben, gibt EOF ein TRUE zurück, wenn das letzte GET keinen vollständigen Record mehr lesen kann und versucht hinter dem Dateiende zu lesen. Der Rückgabewert ist eine Integer-Zahl.

• Siehe auch

`OPEN, LOF, LOC`

• Beispiel

```
OPEN "TEST.DAT" FOR INPUT AS #1
laenge=0
WHILE NOT EOF(1)
   lese$=INPUT$(1,#1)
   laenge=laenge+1
WEND
CLOSE #1
PRINT laenge   'Die Länge der Fatei in Bytes. Man
               'kann stattdessen auch LOF nehmen.
```

ERASE Befehl

• Syntax

ERASE *Array_Name* [, *Array_Name*]...

• Wirkungsweise

Dieser Befehl löscht den Inhalt dynamischer Arrays, die zuvor mit DIM dimensioniert wurden.

• Details

Wurde ein Array erst einmal mit ERASE gelöscht, kann es erst wieder benutzt werden, nachdem es mit DIM wieder dimensioniert wurde. Nur mit REDIM allein kann man ein Array löschen und zugleich wieder neu dimensionieren. Bei REDIM PRESERVE erlaubt der Zusatz PRESERVE, daß ein Array erweitert oder gekürzt werden kann, ohne daß der Inhalt verlorengeht. Mit diesem Befehl lassen sich bereits vorhandene Arrays in den Speicher schreiben. Man sollte deswegen extrem vorsichtig mit ihm umgehen, wenn man ihn in Funktionen oder Unterprogrammen benutzt, die mit Arrays oder Array-Elementen als variable Parameter aufgerufen werden.
Es können nur dynamische Arrays mit ERASE gelöscht werden.

• Siehe auch

DIM, REDIM

• Beispiel

```
REM $dynamic
DIM Arr(15000)

' irgendwelche Aktionen mit Arr()

ERASE Arr   ' löscht den Platz, der von Arr()
            ' belegt wurde
```

ERL, ERR Funktions

• Syntax

```
ERL
ERR
```

• Wirkungsweise

Diese Funktionen geben den Fehlerstatus des aktuellen Programms zurück.

• Details

In einer Fehlerbehandlungsroutine gibt ERR die Fehler- und ERL die Zeilennummer, in der der Fehler aufgetaucht ist, zurück. Wenn ein Fehler in einer Zeile ohne Zeilennummer auftaucht, gibt ERL die Nummer der letzten Zeile mit Nummer zurück.
Wenn man überhaupt keine Zeilennummern verwendet, gibt ERL die physikalische Zeilennummer zurück, in der der Fehler aufgetaucht ist. Das ist die Zeilennummer, die sich ergibt, wenn der Compiler alle Include-Dateien eingebunden hat. Dieses Verhalten findet man nur in MaxonBASIC.

• Siehe auch

```
ON ERROR, ERROR.
```

• Beispiele

```
IF ERR=53 THEN
  PRINT "Datei wurde nicht gefunden in oder nach
        Zeile ";ERL
END IF
.

99 PRINT "Ende"
   RETURN

  I=0
  PRINT 1/I
'Nach dem Ausführen von 1/I ist der Wert von ERL
'99, obwohl diese Zeilennummer nichts mit dem
'ausgeführten Code zu tun hat.
```

ERROR Befehl

• Syntax

```
ERROR Integer_Ausdruck
```

• Wirkungsweise

Dieser Befehl simuliert einen Fehler des laufenden BASIC-Programms.

• Details

ERROR bildet das Gegenstück zu der Funktion ERR, bei der der *Integer_Ausdruck* die Fehlernummer ist, die zurückgegeben wird. Man kann sogar seine eigenen Fehlernummern kreieren, indem man Nummern nimmt, die noch nicht in MaxonBASIC vorkommen.

• Siehe auch

ERL, ERR, ON ERROR.

• Beispiel

```
IF Tst <= 0 THEN ERROR 99 'Noch nicht
                          'reservierte Fehlernummer
IF ERR = 99 THEN
   PRINT "Da ging etwas schrecklich schief!"
END IF
```

EXIT Befehl

• Syntax

```
EXIT {DEF|DO|FUNCTION|IF|FOR|LOOP|SELECT|SUB|
      Bezeichner}
```

• Wirkungsweise

Beendet eine Funktionsdefinition, ein BASIC-Unterprogramm oder einen strukturierten Befehl (z.B. Schleife).

• Details

EXIT DEF beendet eine Funktionsdefinition, EXIT SUB ein Unterprogramm. EXIT DO, EXIT IF, EXIT FOR, EXIT FUNCTION, EXIT LOOP und EXIT SELECT bewirken, daß der strukturierte Befehl, auf den sie sich beziehen, sofort verlassen wird. Die Programmausführung geht dann nach dem END weiter, auf das sich der Befehl bezogen hat. EXIT *Bezeichner* sorgt dafür, daß die durch den *Bezeichner* benannte REPEAT-Schleife frühzeitig abgebrochen wird.
Diese Befehle werden zum Verlassen von Routinen angewandt (z.B. als ein Teil eines IF...THEN...END IF-Blocks). Sie entsprechen nicht dem Ende der Routinendefinition; END DEF, END FUNCTION, END SUB und END REPEAT *Bezeichner* sind dafür erforderlich.

• Siehe auch

DO, IF, FOR, FUNCTION, LOOP, SELECT, SUB

• Beispiel

```
SUB Fred
.
.
IF Var&=12654 THEN EXIT SUB
.
.
END SUB
FOR i=1 TO max
  IF a(i)=0 THEN EXIT FOR
NEXT i
IF i>max THEN
  PRINT "Kein Element mit dem Wert 0 vorhanden!"
ELSE
  PRINT "Erstes Element mit dem Wert 0 ist ";i
END IF
```

EXP Funktion

• Syntax

EXP(*numerischer_Ausdruck*)

• Wirkungsweise

Berechnet die Exponentialfunktion eines *numerischen_Ausdrucks*.

• Details

Diese Funktion gibt die Potenz eines *numerischen_Ausdrucks* zur Basis *e* (natürlicher Logarithmus, auch Eulersche Zahl) zurück. Wenn der *numerische_Ausdruck* über eine doppelte Genauigkeit verfügt, wird das Ergebnis auch wieder doppelt genau zurückgegeben, ansonsten einfach genau.

• Siehe auch

LOG

• Beispiel

```
PRINT EXP(1#)
```

Ergebnis:

```
2.718281828459046
```

FEXISTS Funktion

• Syntax

```
FEXISTS(Dateiname)
```

• Wirkungsweise

Überprüft, ob eine Datei existiert oder nicht.

• Details

Dateiname muß den Regeln des AmigaDOS entsprechen. Es muß der vollständige Pfad angegeben werden. Die Funktion gibt eine –1 zurück, wenn die Datei gefunden wurde, eine 0, wenn nicht.

• Siehe auch

OPEN.

• Beispiel

```
f$="TEST.DAT"
IF NOT FEXISTS(f$) THEN
  f$="WORK:"+f$
  IF NOT FEXISTS(f$) THEN
       do_error "Datei nicht gefunden!"
  END IF
END IF
OPEN f$ FOR INPUT AS #1
```

FIELD Befehl

• Syntax

```
FIELD [#]Kanalnummer, Feldbreite AS
      String_Variable...
```

• Wirkungsweise

Dieser Befehl reserviert Speicher für Variablen in dem Puffer für wahlfreien Zugriff (Random Access Buffer).

• Details

Die *Kanalnummer* ist die Nummer, mit der im OPEN-Befehl die Datei geöffnet wurde. *Feldbreite* ist die Speicherplatzmenge in Zeichen, die die *String_Variable* benötigt.
Die gesamte Anzahl an Bytes darf nicht die Record-Länge überschreiten, die beim Öffnen der Datei angegeben wurde. Die Grundeinstellung beträgt 128 Bytes.
Wenn eine Datei mit CLOSE geschlossen wird, gehen alle Felddefinitionen verloren.
Der FIELD-Befehl liest keine Daten in den Puffer, er reserviert nur den Platz darin für die Variable.
Achtung: Man sollte keine Feldvariable in einem INPUT-Befehl benutzen, wenn sie eine Feldvariable bleiben soll. Ein INPUT-Befehl oder eine andere Zuweisung bewirkt, daß der Variablenzeiger auf den Platz des Strings und nicht auf den Puffer für wahlfreien Zugriff zeigt. Man sollte LSET und RSET benutzen, um Feldvariablen zuzuweisen.

• Siehe auch

OPEN, GET, PUT, LSET, RSET

• Beispiel

```
'Die erste Definition alloziert den 19-Byte-
'Puffer für das Land, die Stadt und Nummer; die
'zweite den ganzen Puffer für die einzelne
'Variable Ganze_Nummer$.
OPEN "TELEFON.DAT" FOR INPUT AS #1 LEN=19
FIELD #1, 4 AS Land$, 5 AS Stadt$, 10 AS Nummer$
FIELD #1, 19 AS Ganze_Nummer$
```

FILES Befehl

• Syntax

`FILES [Dateispezifikationen]`

• Wirkungsweise

Dieser Befehl listet die Dateinamen eines bestimmten Verzeichnisses (Directory) auf.

• Details

Dateispezifikationen muß ein String sein, der den Regeln des AmigaDOS entspricht. Er kann einen Geräte- oder Laufwerksnamen und/oder einen Pfadnamen enthalten. Directories werden in eckigen Klammern ([ ]) angezeigt.
Platzhalter (Wildcards) werden *nicht* unterstützt.

• Beispiel

```
FILES "Maxon:" 'Dies gibt eine Liste aller
               'Dateien im Verzeichnis
               'Maxon: aus.
```

FIX Funktion

• Syntax

`FIX(numersicher_Ausdruck)`

• Wirkungsweise

Über diese Funktion erhält man den abgeschnittenen Integer-Teil eines *numerischen Ausdrucks.*

• Details

Der Unterschied zwischen FIX und INT besteht darin, daß FIX, wenn der *numerische_Ausdruck* negativ ist, die erste negative Integer-Zahl liefert, die größer als sie selbst ist. INT gibt die erste negative Integer-Zahl zurück, die kleiner als der *numerische_Ausdruck* ist. Der Zahlentyp der Ergebnisses entspricht dem Zahlentyp des *numerischen_Ausdrucks.*

• Siehe auch

`INT, CINT, CLNG, CDBL, CSNG`

• Beispiel

```
PRINT FIX(1.5)
PRINT FIX(-1.5)
```

Ergebnis:

```
 1
-1
```

FOR...NEXT Befehl

• Syntax

```
FOR zaehler = start TO ende [STEP inkrement]
.
.
NEXT [zaehler][,zaehler]...
```

• Wirkungsweise

Dieser Befehl führt eine Reihe von Anweisungen in einer Schleife aus.

• Details

Alle vier Werte können entweder Integer- oder Long-Integer-Zahlen bzw. Fließkommanzahlen mit einfacher oder doppelter Genauigkeit sein. Die Programmzeilen zwischen FOR und NEXT werden ständig wiederholt. Nach jedem Durchlauf der Schleife wird der *zaehler* hochgezählt; dabei kann mit *inkrement* eine Schrittweite angegeben werden. Der *zaehler* wird mit dem *ende*-Parameter verglichen. Wenn der *zaehler* größer als *ende* ist, wird das Programm mit der nächsten Programmzeile nach NEXT fortgeführt.
Der *zaehler* wird vor dem Vergleich hochgezählt.
Wenn kein *inkrement* angegeben wird, hat die Schrittweite von *inkrement* automatisch den Wert 1. Wenn *inkrement* negativ ist, muß *ende* kleiner als *start* sein, anderenfalls wird die Schleife nicht ausgeführt.
Wenn *start* kleiner als *ende* ist und *inkrement* ist positiv, wird die Schleife ebenfalls nicht ausgeführt.
FOR...NEXT-Schleifen dürfen ineinander verschachtelt werden, vorausgesetzt jede Schleife hat ihren eigenen *zaehler*. Der NEXT-Befehl der inneren Schleife muß vor dem NEXT-Befehl der äußeren Schleife stehen.
Der NEXT-Befehl mit dem Format *NEXT x,y,z* entspricht

```
            NEXT x
      NEXT y
NEXT z
```

Wird beim NEXT-Befehl kein *zaehler* angegeben, bezieht sich NEXT auf den letzten vorangegangenen FOR-Befehl.
FOR...NEXT-Schleifen können mit EXIT- oder GOTO-Befehlen vorzeitig abgebrochen werden.

Achtung: Die Summe aus *ende* und *inkrement* (auch wenn *inkrement* nicht angegeben wurde, und somit den Wert 1 hat) darf niemals den in der Variable *zaehler* angegebenen Bereich überschreiten, anderenfalls tritt ein Overflow-Fehler auf.
Optimierung: Eine FOR...NEXT-Schleife wird dann am schnellsten ausgeführt, wenn der *zaehler* eine Integer-Zahl und *start*, *ende* und *inkrement* Konstanten sind. Wenn bei den drei zuletzt genannten Variablen verwendet werden müssen, sollten diese am besten vom Typ Integer sein, um eine maximale Geschwindigkeit zu erreichen.

• Siehe auch

WHILE, DO...LOOP.

• Beispiel

```
DEFINT I
FOR I_EINS = 1 TO 5
   FOR I_ZWEI = 3 TO 11 STEP 2
        FOR I_DREI = 0 TO -15 STEP -1
.
.
NEXT I_DREI,I_ZWEI,I_EINS
```

FORMATI$, FORMATL$, FORMATS$ und FORMATD$ Funktionen

• Syntax

```
FORMATI$(Integer_Ausdruck, String_Ausdruck)
FORMATL$(Long-Integer_Ausdruck, String_Ausdruck)
FORMATS$(Einfach_Ausdruck, String_Ausdruck)
FORMATD$(Doppelt_Ausdruck, String_Ausdruck)
```

• Wirkungsweise

Diese Funktionen formatieren numerische Ausdrücke, ähnlich wie es die Befehle PRINT USING und PRINT machen. Das Ergebnis liegt allerdings hier als String vor.

• Details

Diese Funktionen kann man dazu benutzen, Zahlen in Strings zu konvertieren. Wenn der *String_Ausdruck* leer ist (""), erhält man den String in der Form, wie er beim einem normalen PRINT-Befehl ausgegeben würde; ansonsten wird der *String_Ausdruck* wie bei einem PRINT USING-Format-String ausgegeben. Weitere Informationen zu den Formatiermöglichkeiten findet man unter dem PRINT USING-Befehl. Die Funktionen sind vor allem sehr nützlich, wenn man direkt mit dem Betriebssystem drucken will.

• Siehe auch

PRINT USING, PRINT, STR$.

• Beispiel

```
WRITE FORMATI$(12,""),FORMATI$(12,"###")
WRITE FORMATS$(1E10,""),FORMATS$(1E10,
                          "##.##^^^^")
WRITE FORMATD$(1E10,""),FORMATD$(1E10,
                          "##.##^^^^")
```

Ergebnis:

```
" 12 "," 12"
" 1E+10 "," 1.00E+10"
" 10000000000 "," 1.00D+10"
```

FRE Funktion

• Syntax

FRE(*numerischer_Ausdruck*)
FRE(*String_Ausdruck*)

• Wirkungsweise

Diese Funktion gibt entweder den freien Speicher des AmigaDOS oder die Größe des Systemheaps zurück.

• Details

Verschiedene *numerische_Ausdrücke* führen zu unterschiedlichen Resultaten:

-1 Anzahl der freien Bytes im System
-2 Anzahl der freien Bytes des Stacks, die nicht in Benutzung sind.
-3 liefert den größten freien Speicherblock
jeder andere Wert liefert den freien Platz des BASIC-Heaps

Bei einem *String_Ausdruck* wird die Größe des zusammenhängenden freien Heap-Speichers nach einer Garbage Collection in Bytes zurückgegeben.
Unter dem *Heap* versteht man den Speicherbereich, der vom BASIC für die Verwaltung der String-Variablen und Arrays benötigt wird.
Wenn man die HEAPDYNAMIC-Option benutzt, erhält man den maximal für das Programm zur Verfügung stehenden Speicher mit FRE("")+FRE(-1).

• Beispiel

```
PRINT FRE(0)
PRINT FRE("")     'Das ist ein Leer-String
PRINT FRE(-1)
```

Ergebnis:

```
 127534
 154293
 4096
```

FREEFILE Funktion

• Syntax

```
FREEFILE
```

• Wirkungsweise

Diese Funktion gibt die erste freie Dateinummer als Integer-Zahl zurück.

• Details

FREEFILE kann in Unterprogrammen dazu genutzt werden, um eine frei Dateinummer für die Ein-/Ausgabe zu erhalten. So kann man leicht feststellen, welche Nummer noch nicht in Gebrauch ist.

• Siehe auch

```
OPEN, CLOSE
```

• Beispiel

```
'Diese Routine zeigt eine Textdatei ohne
'Unterbrechung auf dem Bildschirm an.
SUB ZeigeDatei(dateiname$)
STATIC meinedatei,temp$
meinedatei=FREEFILE
OPEN dateiname$ FOR INPUT AS #meinedatei
WHILE NOT EOF(meinedatei)
   LINE INPUT #meinedatei,temp$
   PRINT temp$
WEND
CLOSE meinedatei
END SUB
```

FUNCTION...END FUNCTION Befehl

• Syntax

```
FUNCTION global_Name[(Parameterliste)]
.
.
[EXIT FUNCTION]
[global_Name=Ausdruck]
.
END FUNCTION
```

• Wirkungsweise

Mit diesem Befehl definiert man eine Funktion.

• Details

Dieser Befehl definiert eine Funktion, ähnlich wie man ein Unterprogramm definiert.
global_Name ist der Name der Funktion. Er darf nur für die Funktion verwendet werden, sonst nirgendwo im Programm.
Die *Parameterliste* stellt die Liste dar, die die Parameter enthält, die an ein Unterprogramm übergeben werden. Dabei kann es sich um Variablennamen, Array-Namen, die mit () beendet sind, oder einfach Zahlenwerte handeln. Wenn BYVAL benutzt wird, sind die Parameter immer Zahlenwerte; dadurch wird die Ausführung des Programmes beschleunigt.
EXIT FUNCTION bewirkt, daß das Programm in das Unterprogramm zurückkehrt. Man benutzt diesen Befehl, um die Funktion bereits vor dem END FUNCTION-Befehl zu verlassen.
Variablen in einem Unterprogramm können LOCAL, SHARED oder STATIC deklariert werden.
Wenn man eine Funktion aufrufen will, bevor sie definiert wurde, muß man den DECLARE-Befehl benutzen.

• Siehe auch

SUB, DECLARE, STATIC, SHARED, LOCAL, EXIT, END, CALL

• Beispiel

```
FUNCTION langhex$(BYVAL a&)
STATIC a$
a$=HEX$(a&)
IF LEN(a$)<8 THEN
   a$=STRING$(8-LEN(a$),"0")+a$
END IF
langhex$=a$
END FUNCTION

PRINT langhex$(42)
```

GET Dateiein-/ausgabebefehl

• Syntax

```
GET [#]Kanalnummer[,Record_Nummer]
```

• Wirkungsweise

Dieser Befehl liest einen Record aus einer Datei mit wahlfreiem Zugriff (Random Access) in den Random-Access-Dateipuffer.

• Details

Bei der *Kanalnummer* handelt es sich um die Nummer, unter der die Datei geöffnet wurde. Wenn die *Record-Nummer* weggelassen wird, wird der nächste Record nach dem letzten GET gelesen.
Um sicherzugehen, daß man nicht versucht hinter dem Dateiende zu lesen, sollte man EOF benutzen.

• Siehe auch

OPEN, FIELD, PUT, INPUT#, Kapitel 6: Konzepte.

• Beispiel

```
'Diese Routine gibt die Namen und das Alter der
'ersten zehn Leute aus, die in der Datei
'ALTER.DAT gelistet sind.
DEFINT I
OPEN "ALTER.DAT" FOR RANDOM AS #1
FIELD #1, 10 AS Dateiname$, 10 AS Leutename$,
          2 AS Alter$
FOR I = 1 TO 10
  GET #1,I 'liest die ersten zehn Einträge
  PRINT Dateiname$;" ";Leutename$;" ist ";
        Alter$;" Jahre alt."
NEXT I
CLOSE #1
```

GET Grafikbefehl

• Syntax

GET(*x1,y1*)-(*x2,y2*),*Array_Name*[(*Index...*)]

• Wirkungsweise

Dieser Befehl speichert Bilder bzw. Bildausschnitte vom Bildschirm.

• Details

GET speichert einen rechteckigen Bildausschnitt, der mit (*x1,y1*) - (*x2,y2*) definiert ist, in das Array mit dem Namen *Array_Name*. Das Gegenstück zu diesem Befehl findet man in PUT; damit gibt man das Bild wieder auf dem Bildschirm aus.
(*x1,y1*) - (*x2,y2*) sind die diagonal gegenüberliegenden Eckpunkte des Rechtecks auf dem Bildschirm. Man muß darauf achten, daß die Koordinaten relativ zur linken oberen Ecke des Bildschirms stehen, nicht zum aktuellen Fenster – im Gegensatz zu vielen anderen Grafikbefehlen. Der Speicherbedarf eines Arrays in Bytes zum Speichern eines Bildes errechnet sich folgendermaßen:

```
2*ebenen*(y2-y1+1)*[(x2-x1)\16+1] + 6
```

ebenen ist die Anzahl der Ebenen des Bildschirmspeichers. Um beispielsweise ein Bild mit den Maßen (10,20) - (100,200) eines 4-Farb-Bildschirms zu speichern, benötigt man 4350 Bytes (2*2*181*6+6) oder ein Integer-Array von 2175 Elementen.
Wenn man ein Integer-Array *ebenen%()* im GET-Befehl benutzt, können multi-dimensionale Arrays verwandt werden; das erleichtert z.B. das Umschalten zwischen verschiedenen Bildern. Maxon BASIC speichert Arrays allerdings in entgegengesetzter Reihenfolge zu AmigaBASIC; man muß also ggf. seinen Code anpassen, wenn man etwas übernehmen will.

• Siehe auch

PUT

• Beispiel

```
DIM p%(2174)
AREA 30,30: AREA 30,60: AREA 60,30: AREAFILL
GET (10,20)-(100,200),p%
PUT (30,40),p%
```

GOSUB...RETURN Befehl

• Syntax

```
GOSUB {Zeilennummer1|Zeilenlabel1}
.
.
RETURN {Zeilennummer2|Zeilenlabel2}
```

• Wirkungsweise

Dieser Befehl verzweigt zu einer Unterroutine und kehrt anschließend wieder zurück.

• Details

Zeilennummer1 oder *Zeilenlabel1* geben die erste Zeile der Unterroutine an, zu der GOSUB verzweigt. Man kann von der Unterroutine auch an *Zeilennummer2* oder *Zeilenlabel2* zurückkehren; es ist also nicht notwendig das Programm genau hinter dem GOSUB-Aufruf fortzuführen. Diese Methode ist allerdings nicht sehr empfehlenswert, da man so einen sehr unübersichtlichen Quelltext bekommt.
Eine Unterroutine kann mehr als einen RETURN-Befehl enthalten. Ein RETURN ohne *Zeilennummer2* oder *Zeilenlabel2* verzweigt zurück zu der Zeile nach dem GOSUB, mit dem die Unterroutine aufgerufen wurde.
Achtung: Wir empfehlen, lieber Unterroutinen mit CALL und SUB...END als mit GOSUB...RETURN zu benutzen. SUB...END-Routinen sind wesentlich flexibler, da man ihnen Parameter übergeben kann und sie lokale Variablen unterstützen.
MaxonBASIC unterstützt keine GOSUB-Aufrufe zu einem Unterprogramm.

• Siehe auch

SUB, FUNCTION

• *Beispiel*

```
I% = 0
PRINT "Hauptschleife"
GOSUB Schleife1
Zurueck_ins_Hauptprogramm:
PRINT "Zurück in der Hauptschleife"
I% = 1
GOSUB Schleife2
PRINT "Fertig"
END
.
.
Schleife1:
PRINT "Erste Unterroutine"
RETURN Schleife2
Schleife2:
PRINT "Zweite Unterroutine"
  IF I% = 0 THEN
       RETURN Zurueck_ins_Hauptprogramm
  ELSE
       RETURN
  END IF
```

Ergebnis:

```
Hauptschleife
Erste Unterroutine
Zweite Unterroutine
Zurück in der Hauptschleife
Zweite Unterroutine
Fertig
```

GOTO Befehl

• Syntax

```
GOTO {Zeilennummer|Zeilenlabel}
```

• Wirkungsweise

Dieser Befehl bewirkt einen unbedingten Sprung im Programm, d.h. daß die Programmausführung an einer bestimmten Stelle fortgeführt wird, ohne daß eine Bedingung abgefragt wird.

• Details

GOTOs dürfen nicht zum Sprung in Funktionen, Unterprogramme oder Unterroutinen verwendet werden; das gleiche gilt für das Verlassen dieser. GOTOs dürfen in Strukturen benutzt werden, sofern dies notwendig ist.

Achtung: Wir empfehlen, lieber strukturierende Befehle, wie IF...THEN...ELSE und DO...LOOP, zwecks besserer Lesbarkeit des Quelltextes und leichterer Fehlersuche zu verwenden. Benötigt man trotzdem ein GOTO, sollte man lieber ein Zeilenlabel angeben, als zu einer Zeilennummer zu springen.

• Siehe auch

```
IF, DO, EXIT
```

• Beispiel

```
GOTO Label
'Hierher kommt das Programm nie!
.
.
Label:
PRINT "Hier geht's weiter!"
```

HEX$ Funktion

• Syntax

HEX$(*numerischer_Ausdruck*)

• Wirkungsweise

Diese Funktion gibt den hexadezimalen Wert eines *numerischen Ausdrucks* in Form eines Strings zurück.

• Details

Der *numerische Ausdruck* wird gerundet, bevor er von HEX$ umgewandelt wird. Wenn es sich bei dem Parameter um eine Integer-Zahl handelt, hat der Ergebnis-String eine Länge von 1 bis 4 Zeichen, bei einer Long-Integer-Zahlen bis zu 8 Zeichen.

• Siehe auch

OCT$, BIN$.

• Beispiel

```
'Diese Routine gibt die Zahlen 1 bis 15 in
'hexadezimaler Schreibweise aus.
DEFINT I
FOR I=1 TO 15
   PRINT HEX$(I);
NEXT I
```

Ergebnis:

```
123456789ABCDEF
```

IF...THEN...ELSE Befehl

• Syntax

```
IF Boolscher_Ausdruck THEN
   Befehl1
   [Befehl2]
.
.
[ELSEIF Boolscher_Ausdruck THEN
   Befehl3
   [Befehl4]]
.
.
[ELSE
   Befehl5
   [Befehl6]]
.
.
END IF
```

oder

```
IF Boolscher_Ausdruck THEN Befehl1
                                    [ELSE Befehl2]
```

• Wirkungsweise

Diese Befehlsstruktur erlaubt eine bedingte Ausführung einer Befehlsfolge oder ein Verzweigen. Für die Bedingung wird ein *boolscher Ausdruck* ausgewertet.

• Details

In der einzeiligen Variante von IF...THEN...ELSE, wird *Befehl1* dann ausgeführt, wenn der *boolsche Ausdruck* TRUE ist, ist er FALSE wird *Befehl2* ausgeführt. Wird kein ELSE angegeben und der *boolsche Ausdruck* ist FALSE, wird das Programm in der Zeile nach dem IF...THEN-Befehl fortgesetzt.
Stehen weitere Befehle hinter dem THEN, sollte man die einzeilige Variante bevorzugen; END IF ist dann nicht erforderlich.
Folgende Programmzeilen sind nicht erlaubt:

```
IF a THEN PRINT "line 1"
   PRINT "line 2"
END IF
```

wogegen

```
IF a THEN
   PRINT "line 1"
   PRINT "line 2"
END IF
```

korrekt ist.

• Siehe auch

SELECT.

• Beispiel

```
S!=RND
IF S!<0.5 THEN
   PRINT "S! ist kleiner als 0.5"
ELSEIF S!>0.5 THEN
   PRINT "S! ist groesser als 0.5"
ELSE
   PRINT "S! ist gleich mit 0.5"
END IF
```

INCR Befehl

• Syntax

`INCR numerische_Variable`

• Wirkungsweise

Dieser Befehl erhöht eine *numerische_Variable* um Eins.

• Details

Bei der *numerischen_Variablen* kann es sich um eine einfache oder auch eine Array-Variable jeglichen numerischen Typs handeln, die dann um Eins erhöht wird.

• Siehe auch

`DECR`.

• Beispiel

```
INCR a(i,j) 'entspricht a(i,j)=a(i,j)+1
```

INITHOOK Befehl

• Syntax

`INITHOOK` *`Hook_Adresse, Funktion_Adresse`*

• Wirkungsweise

Dieser Befehl initialisiert einen „Hook", damit er vom Betriebssystem aufgerufen werden kann.

• Details

Einen Hook benötigt man beim Amiga dann, wenn man eigene Routinen ins Betriebssystem einbinden möchte. INITHOOK initialisiert einen Hook, so daß die korrekte Zusammenarbeit mit den Register-beschreibungen von MaxonBASIC gewährleistet ist.
Beide Parameter dieses Befehls sind Long-Integer-Zahlen. Der erste enthält die Speicheradresse des Hooks an, der zweite die Adresse einer Funktion oder eines Unterprogramms mit drei Long-Integer-Parametern, wie weiter unten im Beispiel beschrieben.

• Beispiel

```
DEFINT a-z
REM $include utility.bh
LIBRARY OPEN "utility"
FUNCTION meineHookFunktion&(BYVAL Hook&,
                            BYVAL o&,BYVAL msg&)
PRINT PEEK$(o&)
meineHookFunktion&=42
END FUNCTION

DIM h(Hook_sizeof\2)
INITHOOK VARPTR(h(0)),VARPTRS(meineHookFunktion&)
res&=CallHookPkt&(VARPTR(h(0)),SADD("Hallo"+CHR$(0)),0)
PRINT "gab die Funktion zurueck";1
```

INKEY$ Funktion

• Syntax

```
INKEY$
```

• Wirkungsweise

Diese Funktion liest ein Zeichen – sofern eines vorhanden ist – von der Tastatur ein, ohne daß dieses auf dem Bildschirm angezeigt wird.
INKEY$ gibt entweder einen Leer-String oder einen 1-Byte-String mit dem Zeichen, das eingelesen wurde, zurück.
Einen Leer-String ("") erhält man dann, wenn keine Taste gedrückt wurde.
Einen 1-Byte-String erhält man dann, wenn eine normale Taste gedrückt wurde. Er enthält den ASCII-Wert des Zeichens. Folgende Tasten geben Spezialwerte zurück:

Backspace	8	F1	128
Return	13	F2	129
↑	28	F3	130
↓	29	F4	131
→	30	F5	132
←	31	F6	133
Del	127	F7	134
Help	139	F8	135
		F9	136
		F10	137

Achtung: Die Amiga-Tasten geben keinen Wert zurück, wenn sie mit INKEY$ abgefragt werden.

• *Beispiel*

```
'Diese Beispiel zeigt wie man eine schnelle
'Abfrage machen kann, wenn man nur ein Zeichen
'benötigt.
PRINT "Wählen Sie (1 oder 2)"
DO
  a$=INKEY$
  SELECT CASE a$
       CASE ="1"
            CALL Eins: EXIT LOOP
       CASE ="2"
            CALL Zwei: EXIT LOOP
  END SELECT
LOOP
```

INPUT Befehl

• Syntax

```
INPUT [;]["Text"{;|,}] Variablenliste
```

• Wirkungsweise

Dieser Befehl wartet auf eine Eingabe des Anwenders, die dann der *Variablenliste* zugewiesen wird.

• Details

INPUT bewirkt, daß die Programmausführung angehalten und auf die Eingabe des Anwenders gewartet wird. Wenn ein *Text* angegeben wird, wird dieser zuerst ausgegeben und anschließend ein Fragezeichen. Wenn man auf das Fragezeichen verzichten will, muß hinter dem *Text* ein Komma folgen.
Wenn man ein Fenster benutzt, kann man erreichen, daß der Cursor in der selben Zeile bleibt, indem man ein Semikolon hinter den INPUT-Befehl schreibt – sogar nach Drücken der Enter-Taste. Aufgrund der Art und Weise, wie AmigaDOS Eingaben puffert, ist diese Option bei einer Konsoleneingabe nicht verfügbar.

• Siehe auch

```
INKEY$, PRINT
```

• Beispiel

```
INPUT "Das Quadrat von ",Val%
PRINT " ist ";(Val%*Val%)
```

Ergebnis:

```
Das Quadrat von 2 ist 4
```

INPUT# Befehl

• Syntax

```
INPUT #Kanalnummer,Variablenliste
```

• Wirkungsweise

Dieser Befehl liest Daten von einem Gerät oder einer Datei, die durch eine *Kanalnummer* definiert werden und weist sie einer *Variablenliste* zu.

• Details

Bei der *Kanalnummer* handelt es sich um die Nummer, die beim Öffnen einer Datei angegeben werden muß.
Die zu lesenden Daten müssen numerisch sein, wenn sie in numerischen Variablen eingelesen werden sollen. Führende Carriage-Returns, Line-Feeds und Leerstellen werden ignoriert; das erste Zeichen, das nicht einem dieser drei Typen entspricht, wird als Beginn einer Zahl interpretiert. Eine Zahl muß durch ein Carriage-Return, Line-Feeds oder Leerstellen beendet werden. Wird das Ende der Datei (EOF) während des Lesens eines Wertes erreicht, wird die Variable beendet.
In der Grundeinstellung beträgt die maximale Länge einer Variable, die mit diesem Befehl eingelesen werden kann, 1024 Bytes, was für die meisten Anwendungen ausreichen dürfte. Man kann diesen Wert allerdings erhöhen, indem man den Langwert an der Adresse SYSTAB+40 verändert.

• Siehe auch

```
BLOAD, INPUT$, PRINT#,SYSTAB
```

• Beispiel

```
INPUT #1,Strin$,Int%,Lng_int&,Sing!,Dop_genau#
```

INPUT$ Befehl

• Syntax

```
INPUT$(n[,[#]Kanalnummer])
```

• Wirkungsweise

Dieser Befehl liest *n* Zeichen von einem angegebenen Kanal.

• Details

Wenn die *Kanalnummer* weggelassen wird, werden die Zeichen direkt von der Tastatur gelesen, ohne das die Zeichen auf dem Bildschirm angezeigt werden. Der Befehl funktioniert dann so ähnlich wie der INKEY$-Befehl, allerdings mit einer durch *n* genau bezifferten Anzahl von zu lesenden Zeichen. *n* kann eine Integer- oder Long-Integer-Zahl sein. Insofern genug Speicher vorhanden ist, kann mit INPUT$ auch eine ganze Datei auf einmal eingelesen werden.

• Siehe auch

BLOAD, INPUT, PRINT#.

• Beispiel

```
'Dieses Programm liest die ersten 512 Bytes der
'Datei "a.daten" in die String-Variable Daten$
OPEN "a.daten" FOR INPUT AS #1
Daten$=INPUT$(512,#1))
CLOSE #1
```

INSTR Funktion

• Syntax

`INSTR([Start,]String1,String2)`

• Wirkungsweise

Diese Funktion sucht vorwärtsgerichtet, ob *String2* in *String1* vorhanden ist.

• Details

Bei *Start* handelt es sich um einen optionalen Offset-Wert, ab dem in *String1* gesucht werden soll. *Start* kann entweder eine Integer- oder Long-Integer-Zahl sein. Die Strings können entweder String-Variablen, Buchstaben-Strings oder String-Ausdrücke sein. Der Wert, der zurückgegeben wird, beinhaltet die Position des ersten Zeichens des gefundenen Strings.
Wenn *Start* einen größeren Wert als die Länge von *String1* enthält, wird eine 0 zurückgegeben.
Wenn es sich bei *String1* um einen Leer-String ("") handelt, wird eine 0 zurückgegeben.
Wenn *String2* nicht gefunden wird, wird eine 0 zurückgegeben.
Wenn *String2* ein Leer-String ist, wird *Start* zurückgegeben; ist *Start* nicht angegeben, wird eine 1 zurückgegeben.
Achtung: Da Strings in MaxonBASIC keine Längenbegrenzung haben, gibt diese Funktion eine Long-Integer-Zahl zurück.

• Siehe auch

`RINSTR.`

• Beispiel

```
'Diese Routine durchsucht die Kommandozeile nach
'einer Leerstelle, um den ersten Übergabe-
'parameter zu finden.
I%=INSTR(KOMMANDO$," ")
IF I%=0 THEN I%=LEN(KOMMANDO$)
PRINT "Der erste Übergabeparameter ist: ";
                                LEFT$(KOMMANDO$,I%)
```

INT Funktion

• Syntax

INT(*numerischer_Ausdruck*)

• Wirkungsweise

Diese Funktion gibt die größte Integer-Zahl zurück, die kleiner gleich dem *numerischen_Ausdruck* ist.

• Details

Der Rückgabewert ist vom gleichen Typ wie der *numerische_Ausdruck.*

• Siehe auch

FIX, CINT, CDBL, CSNG, CDBL

• Beispiel

```
'Dieses Beispiel zeigt die Unterschiede zwischen
'den Funktionen CINT, FIX and INT.
PRINT CINT(1.5),CINT(-1.5)
PRINT FIX(1.5),FIX(-1.5)
PRINT INT(1.5),INT(-1.5)
```

Ergebnis:

```
2       -2
1       -1
1       -2
```

KILL Befehl

• Syntax

`KILL` *`Dateispezifikation`*

• Wirkungsweise

Alle Dateien, auf die die *Dateispezifikation* zutrifft, werden vom aktuellen bzw. angegebenen Laufwerk gelöscht.

• Details

Die *Dateispezifikation* muß dem AmigaDOS-Format entsprechen. KILL löscht auch jede Icon(.info)-Datei, auf die die *Dateispezifikation* zutrifft.

• Beispiel

```
KILL "DF0:TEMP"
```

LBOUND Funktion

• Syntax

`LBOUND(Array[,Dimension])`

• Wirkungsweise

Diese Funktion gibt den kleinsten verfügbaren Index eines bestimmten Arrays und optional einer bestimmten Dimension zurück.

• Details

Eine häufige Anwendung von LBOUND ist, zusammen mit UBOUND die Größe eines Arrays zu bestimmen. *Array* ist das Array, das überprüft werden soll, *Dimension* die Nummer der Array-Dimension, die überprüft werden soll.
Da OPTION BASE für alle Indizes des Arrays gilt, haben diese alle dieselbe Untergrenze.

• Siehe auch

`UBOUND, OPTION BASE`

• Beispiel

```
DIM Ebenen(X,Y,Z)
.
.
PRINT LBOUND (Ebenen,2)
'Das Beispiel gibt die Untergrenze für die Y-
'Dimension des Arrays Ebenen zurück; sie sollte
'0 sein, sofern nicht mit OPTION BASE eine 'an-
dere Untergrenze definiert wurde.
```

LCASE$ Funktion

• Syntax

`LCASE$(String_Ausdruck)`

• Wirkungsweise

Diese Funktion gibt einen String in Kleinbuchstaben zurück.

• Details

Jeder Buchstabe, von A bis Z, wird in Kleinbuchstaben (a bis z) gewandelt. Alle anderen Zeichen, auch solche mit Akzenten werden nicht konvertiert.

• Siehe auch

`UCASE$, ASCII-Tabelle`

• Beispiel

```
PRINT "Festplatte formatieren? J/N";
DO a$=INKEY$
UNTIL a$<>""
IF LCASE$(a$)="j" THEN
  do_format "DH0:"
ELSE
  PRINT "Nochmal Glück gehabt!"
END IF
```

LEFT$ Funktion

• Syntax

LEFT$(*String_Ausdruck,n*)

• Wirkungsweise

Diese Funktion schneidet *n* Zeichen eines *String-Ausdrucks* von links ab. Das Ergebnis steht in einem neuen String.

• Details

n ist eine Long-Integer-Zahl, da die Länge eines Strings in MaxonBASIC nicht begrenzt ist. Wenn *n* = 0, wird ein Leer-String zurückgegeben. Ist *n* größer als die Anzahl der Zeichen in dem *String-Ausdruck*, wird der ganze String zurückgegeben.
String-Ausdruck kann eine String-Variable, ein Buchstaben-String oder eine String-Konstante sein.

• Siehe auch

MID$, RIGHT$

• Beispiel

```
String$="MaxonBASIC"
PRINT LEFT$(String$,5)
```

Ergebnis:

```
Maxon
```

LEN Funktion

• Syntax

`LEN(String_Ausdruck)`

• Wirkungsweise

Diese Funktion liefert die Anzahl der Zeichen eines *String-Ausdrucks*.

• Details

String-Ausdruck kann eine String-Variable, ein Buchstaben-String oder eine String-Konstante sein. Der Rückgabewert kann größer als 32767 sein; man sollte also vorsichtig sein, den Rückgabewert als Integer-Zahl zu definieren – lieber Long-Integer-Zahlen verwenden.

• Beispiel

```
String$="MaxonTWIST"
PRINT LEN(String$)
```

Ergebnis:

```
10
```

LET Befehl

• Syntax

`[LET] Variable = Ausdruck`

• Wirkungsweise

Weist einer *Variablen* den Wert eines *Ausdrucks* zu.

• Details

Der LET-Befehl ist optional; es ist völlig ausreichend, einen Wert mittels des Gleichheitszeichens (=) einer *Variablen* zuzuweisen. Will man einer Variablen im Zusammenhang mit dem FIELD-Befehl einen Wert zuweisen, muß man LSET oder RSET verwenden.

• Beispiel

```
LET Profit=Einkommen-Kosten-Steuer
```

entspricht

```
Profit=Einkommen-Kosten-Steuer
```

LIBRARY Befehl

• Syntax

```
LIBRARY Library_Name[,anderer_Library_Name]...
LIBRARY DECLARE Library_Name
                          [,anderer_Library_Name]...
LIBRARY OPEN Library_Name[,Mindestversion]
LIBRARY CLOSE [Library_Name
                          [,anderer_Library_Name]...]
LIBRARY VARPTR Library_Name, Neuer_Zeiger
```

• Wirkungsweise

Dieser Befehl definiert, welche Amiga-Libraries von einem Programm benutzt werden sollen und kontrolliert das Öffnen und Schließen von Amiga-Libraries.

• Details

Das Amiga-Betriebssystem wird in der Regel beim Zugriff auf Libraries aufgerufen; außerdem kann es vorkommen, daß man manchmal Geräte, Ressourcen, Gadget- oder Datentypen ansprechen muß. Sie werden so ähnlich wie richtige Libraries aufgerufen, allerdings unterschiedlich geöffnet.
Um eine Library, ein Gerät, eine Ressource, eine Gadget-Klasse oder einen Datentyp aufzurufen, benötigt man eine *.bmap*-Datei zum *Compilieren* des Programms. Die *.bmap*-Datei enthält Informationen zu den Routinen der Library, wo sich welche Routinen in ihr befinden, wo sie in der Library stehen und welche Parameter in welches Register geschrieben werden. Wenn man ein Programm laufen läßt, werden alle dazugehörenden Dateien (.library , .device, .resource, .gadget oder datatype) benötigt, soweit sie nicht im ROM enthalten sind.
Der Library-Name muß in Anführungszeichen stehen und sollte relativ zu dem LIBS:-Verzeichnis lauten, z.B.:

"graphics.library"
"diskfont.library"
"audio.device"
"gadgets/gradientslider.gadget"
"datatypes/amigaguide.datatype"

Für konventionelle Libraries kann die Endung .library weggelassen werden.

Den Name der .bmap-Datei erhält man, indem man die Endung des Library-Namens (.library,.gadget etc.) abschneidet und sie durch die Endung .bmap ersetzt.
Um mit HiSoft BASIC 1 kompatibel zu sein, wurde folgendes berücksichtigt: Wenn dem Library-Name ein Pfad vorangestellt ist, der entweder einen Doppelpunkt enthält oder mit einem Slash (/) beginnt, wird dieser Pfad auch für die .bmap-Datei benutzt. Er wird allerdings wieder auf den alten Pfad zurückgesetzt, wenn die Library geöffnet wird.
Wenn der Compiler eine .bmap-Datei öffnet, sieht er zuerst im aktuellen Directory und dann unter den Pfaden nach, die mit der LIBPATH-Option angegeben wurden. Findet er dort nichts, sucht er im LIBS:-Verzeichnis. Normalerweise wird der Library-Pfad bei der Installation auf das BMAP:-Verzeichnis gesetzt, wo sich bei MaxonBASIC die .bmap-Dateien für das ganze Betriebssystem befinden.
MaxonBASIC benutzt dasselbe Format für .bmap-Dateien wie AmigaBASIC. Dieses Format kann von FD-Dateien mittels dem FDtoBMAP-Utility erzeugt werden.
Die verschiedenen Anwendungsformen des Library-Befehls sehen folgendermaßen aus:

LIBRARY DECLARE *Library_Name*
[,*anderer_Library_Name*]...

Diese Anwendungsform liest die .bmap-Datei für die angegebenen Libraries, so daß der Compiler den entsprechenden Code generieren kann. Einer dieser Befehle ist in den .bh-Dateien enthalten, die vom Compiler unterstützt werden.

LIBRARY OPEN *Library_Name*[,*Mindestversion*]

Dies öffnet die angegebene Library, wenn diese Zeile während der Laufzeit des Programms ausgeführt wird. Es wird die Anfangsadresse der Library zurückgegeben. Diese wird von dem Code benötigt, der die Library aufruft. Man muß diesen Befehl also vor dem eigentlichen Aufruf ausführen. Wenn eine *Mindestversion* (lauffähig ab Version x) einer Library angegeben wird, wird, sobald nur eine alte Version der Library vorhanden ist, ein Runtime-Error ausgegeben. Dieser Befehl kann nur bei richtigen Libraries, nicht aber bei Geräten oder Datentypen, genutzt werden.

LIBRARY *Library_Name*[,*anderer_Library_Name*]...

Diese Form des LIBRARY-Befehls kombiniert LIBRARY DECLARE und LIBRARY OPEN, überprüft allerdings nicht die Versionsnummer der Library auf eine angegebene Mindestversion. Diese Form dient hauptsächlich der Kompatibilität zu HiSoft BASIC 1 und AmigaBASIC.

LIBRARY CLOSE [Library_Name
[,*anderer_Library_Name*]...]

Dieser Befehl schließt eine angegebene Library oder Libraries. Wenn kein spezieller Name angegeben wird, werden alle offenen Libraries geschlossen. Hat man erst einmal eine Library geschlossen, kann man kein Unterprogramm oder keine Funktion aus der Library aufrufen. Sofern kein anderes Programm dieselbe Library nutzt, wird der Speicher, den die Library belegt hatte, wieder an das Betriebssystem zurückgeben. Wird das BASIC-Programm beendet, werden automatisch alle Libraries geschlossen.

LIBRARY VARPTR Library_Name,neuer_Zeiger

Wenn man eine richtige Amiga-Library aufruft, wird der Zeiger auf die Anfangsadresse der Library, den man zum Aufruf benötigt, durch LIBRARY OPEN gesetzt; für Geräte, Ressourcen oder Datentypen kann man diesen Befehl allerdings nicht benutzen. Stattdessen muß man OpenDevice oder OpenDatatype verwenden, um den korrekten Zeiger zu erhalten. Mit der oben genannten Variante des LIBRARY-Befehls kann man den Zeiger einer angegebenen Library ändern. Man muß darauf achten, daß man den Library-Zeiger für ein Gerät, eine Ressource oder einen Datentyp auf 0 setzt, bevor man das Programm beendet, anderenfalls versuchen die Runtime-Routinen das Gerät, die Ressource oder den Datentyp zu schließen.

• *Siehe auch*

DECLARE, LIBRARY Funktion,

• Beispiel

```
' Diese Aufrufe sind identisch
LIBRARY "diskfont.library"
LIBRARY "diskfont"
LIBRARY DECLARE "diskfont": LIBRARY OPEN
                                          "diskfont"

' öffnet die Workbench 3.0-graphics-Library
LIBRARY "graphics",39

' öffnet die diskfont-Ressource
LIBRARY DECLARE "disk.resource"
diskbase&=OpenResource&(SADD("disk.resource"+CHR$(0))
LIBRARY VARPTR "disk.resource",diskbase&
' jetzt ist die disk-Ressource verfügbar
'...
' schließt die Ressource als eine Library
LIBRARY VARPTR "disk.resource",0
```

LIBRARY Funktion

• Syntax

```
LIBRARY(Library_Name)
```

• Wirkungsweise

Dieser Befehl gibt die Anfangsadresse einer bestimmten Library zurück. Wenn die Library nicht geöffnet ist, wird eine 0 zurückgegeben.

• Details

Der Library-Name muß in Anführungszeichen stehen; es dürfen dabei keine verzweigten Strings benutzt werden. Diese Funktion dient dazu, auf die Datenstruktur des Zeigers, der auf die Anfangsadresse weist, zugreifen zu können.

• Siehe auch

```
LIBRARY Befehl
```

• Beispiel

```
CONST lib_version%=20
LIBRARY "graphics"
grversion=PEEKW(LIBRARY("graphics")+lib_version%)
PRINT "Version";grversion;" von graphics "
```

LINE Befehl

• Syntax

```
LINE [[STEP](x1,y1)] - [STEP](x2,y2)[,Farbe],
                                        [b[f]]
```

• Wirkungsweise

Zeichnet eine Linie, ein Rechteck oder ein gefülltes Rechteck in das aktuelle Fenster.

• Details

Im einfachsten Fall zeichnet dieser Befehl eine Linie zwischen den Punkten (*x1,y1*) und (*x2,y2*). Wenn die Anfangskoordinate (*x1,y1*) weggelassen wird, wird von der aktuellen Grafikposition gezeichnet. Wenn STEP benutzt wird, wird die folgende Koordinate als relativ zur aktuellen Grafikposition, nicht relativ zum aktuellen Fenster, angesehen.

Der *Farbe*-Parameter bestimmt die Farbe, in der die Kinie gezeichnet werden soll. Wird er weggelassen, wird die aktuelle Vordergrundfarbe verwendet (diese kann mit den COLOR-Befehl bestimmt werden). Wenn der letzte Parameter *b* lautet, wird ein Rechteck vom aktuellen Grafikpunkt aus gezeichnet, wobei die Koordinaten die gegenüberliegende Ecke des Rechtecks angeben. Wenn der letzte Parameter *bf* lautet, ist das Rechteck gefüllt.

• Siehe auch

CIRCLE, PATTERN

• Beispiele

```
LINE (100,100) - (150,100)
' zeichnet eine horizontale Linie zwischen den
' Punkten (100,100) und (150,100)

LINE (100,100) - STEP (50,0)
' eine bessere Lösung für obiges Beispiel

LINE -STEP (20,20),2
' zeichnet eine diagonale Linie von der
' aktuellen Grafikposition ausgehend in Farbe 2.

LINE (50,50)- STEP (40,0)
LINE -STEP (0,30)
LINE -STEP(-40,-30)
' zeichnet ein Dreieck
' mit den Eckpunkten (50,50),(90,50),(90,80)

LINE STEP (0,0) - (60,60),,b
' zeichnet eine Box mit der aktuellen Grafik-
' postion als den Eckpunkt und der Koordinate
'(60,60) als anderen.

FOR i=1 TO 3
   LINE (i*20,i*20)- STEP (15,15),i,bf
NEXT i
' zeichnet 3 gefüllte Boxen
' mit unterschiedlichen Farben.
```

LINE INPUT Befehl

• Syntax

```
LINE INPUT[;] ["Text";] String_Variable
```

• Wirkungsweise

Dieser Befehl weist eine ganze Eingabezeile einer *String_Variablen* zu, wobei Trennzeichen, wie z.B. Kommata, ignoriert werden.

• Details

Der *Text* wird ausgegeben und dann auf eine Eingabe gewartet. Ein Fragezeichen wird nicht automatisch mit ausgegeben; es muß, sofern erwünscht, im *Text* stehen. Folgt dem LINE INPUT direkt ein Semikolon, hat der Line-Feed, der das Ende der Eingabezeile markiert, keine Wirkung auf dem Bildschirm.
LINE INPUT benutzt man am besten, wenn man Strings mit Kommata, Fragezeichen oder Leerstellen eingeben will.

• Siehe auch

```
INPUT, READ, PRINT, WRITE
```

• Beispiel

```
LINE INPUT "Geben Sie einen Befehl ein: ",Bef$
```

LINE INPUT# Befehl

• Syntax

```
LINE INPUT #Kanalnummer,String_Variable
```

• Wirkungsweise

Dieser Befehl liest eine Folge von Zeichen, die durch ein Line-Feed beendet ist, von einem Gerät oder einer Datei, die durch eine *Kanalnummer* bestimmt sind, ein und weist sie einer *String_Variablen* zu.

• Details

Dieser Befehl liest eine Folge von Zeichen bis zu einem Line-Feed (inklusive) ein. Hat man mehrere LINE INPUT#-Befehle in Folge, beginnt die zweite Folge direkt hinter dem Line-Feed.
Der LINE INPUT#-Befehl reserviert einen Puffer, wenn er eine Zeile einliest. Die Grundgröße dieses Puffers beträgt 1024 Bytes, folglich lassen sich Zeilen von bis zu 1024 Bytes Größe lesen. Wenn man den Puffer vergrößern will, kann man eine neue Größe (Long-Integer-Zahl) an die Adresse SYSTAB+40 schreiben. Normalerweise ist es allerdings besser und effizienter INPUT$ zu verwenden, wenn man lange Strings einlesen will.

• Beispiel

```
'Diese Routine liest eine Datei, die 5 Records
'(durch Line-Feed abgeschlossen) enthält.
DIM Record$(5)
OPEN "BASIS.LST" FOR INPUT AS #1
I%=0
WHILE NOT EOF(1)
   I%=I%+1
   LINE INPUT #1,Record$(I%)
WEND
CLOSE #1

POKEL SYSTAB+40,2048 ' erhöht den Puffer
                     ' auf 2048 Bytes
```

LOC Funktion

• Syntax

`LOC(Kanalnummer)`

• Wirkungsweise

Diese Funktion gibt dem Programm die aktuelle Position des Dateizeigers einer geöffneten Datei zurück.

• Details

Bei dem Rückgabewert handelt es sich um eine Long-Integer-Zahl. Benutzt man LOC im Zusammenhang mit Dateien mit wahlfreien Zugriff (Random Access), gibt es die Nummer des letzten Records, der gelesen oder geschrieben wurde, zurück. Bei sequentiellen Dateien, die für OUTPUT, INPUT oder APPEND geöffnet wurden, wird die Anzahl der Bytes, die gelesen oder geschrieben wurden, geteilt durch 128 zurückgegeben. Wurden Dateien für COMx: geöffnet, gibt diese Funktion die aktuelle Anzahl der zwischengespeicherten Zeichen im Eingabepuffer zurück.

• Siehe auch

`LOF`

• Beispiel

```
SUB SchreibeBlock
STATIC position%
position%=LOC(2) 'aktuelle Position lesen
LSET a$=MKI$(position%)
PUT #2,1         'In Record 1 steht die letzte
                 'veränderte Record-Nummer
PUT #2,position%  'schreibt den aktuellen Record
END SUB
```

LOCAL Befehl

• Syntax

`LOCAL` *`Variablenliste`*

• Wirkungsweise

Dieser Befehl deklariert Variablen in Funktionen und Unterprogrammen als lokal. Dabei wird jedesmal, wenn die Funktion oder das Unterprogramm verlassen und wieder aufgerufen wird, eine neue Variable erzeugt.

• Details

Dieser Befehl darf nur in Funktionsdefinitionen und Unterprogrammen verwendet werden.
Variablen in DEF FN-Funktionsdefinitionen sind normalerweise global oder STATIC in Unterprogrammen und Funktionen. Der LOCAL-Befehl sorgt dafür, daß jedesmal eine neue Variable erzeugt wird, wenn ein Unterprogramm oder eine Funktion rekursiv aufgerufen werden. Benutzt man den LOCAL-Befehl für Arrays und Dimensionen, braucht das Array nicht mit dem DIM-Befehl dimensioniert werden.

• Beispiel

```
Test=10
PRINT FNbeispiel,Test
DEF FNbeispiel
STATIC Test   ' ohne STATIC würde die globale
Test=5        ' Variable Test hier verändert.
FNbeispiel=1/Test
END SUB
```

LOCATE Befehl

• Syntax

```
LOCATE Zeile[,Spalte [,Cursor]]
```

• Wirkungsweise

Dieser Befehl setzt den Cursor in die angegebene *Spalte* und *Zeile*; außerdem kann der Cursor ein- bzw. ausgeschaltet werden.

• Details

Wenn ein Parameter weggelassen wird, wird der derzeitig gültige Wert von LOCATE nicht verändert.
Bei der *Spalte* handelt es sich um die Nummer der Spalte, in der der Cursor positioniert werden soll. *Zeile* ist die Zeilennummer der Zeile, in der der Cursor positioniert werden soll.
Bei dem *Cursor*-Parameter handelt es sich um einen speziellen MaxonBASIC-Befehl. Wenn mit ihm eine 0 übergeben wird, ist der Cursor ausgeschaltet, bei einer 1 eingeschaltet.

• Siehe auch

```
POS, CSRLIN
```

• Beispiel

```
LOCATE 1,1,0   ' hiermit wird der Cursor in die
               ' linke obere Ecke gesetzt und
               ' gleichzeitig ausgeschaltet.
```

LOF Funktion

• Syntax

`LOF(Kanalnummer)`

• Wirkungsweise

Diese Funktion gibt die Länge der Datei zurück, die durch die *Kanalnummer* bestimmt wurde.

• Details

Der Wert, der für die geöffnete Datei zurückgegeben wurde, ist eine Long-Integer-Zahl.

• Siehe auch

`LOC`.

• Beispiel

```
' Diese Routine öffnet eine Datei, überprüft
' ihre Länge, reserviert genug Speicher und
' liest sie dann vollständig ein.
OPEN "einige.daten" FOR INPUT AS #1
Laenge&=LOF(1)
Dateipuffer$=INPUT$(Laenge&,#1)
CLOSE #1
```

LOG, LOG10, LOG2 Funktionen

• Syntax

```
LOG(numerischer_Ausdruck)
LOG10(numerischer_Ausdruck)
LOG2(numerischer_Ausdruck)
```

• Wirkungsweise

LOG gibt den Natürlichen Logarithmus (zur Basis *e*) eines *numerischen Ausdrucks* zurück.
LOG10 gibt den Logarithmus zur Basis 10 eines *numerischen Ausdrucks* zurück.
LOG2 gibt den Logarithmus zur Basis 2 eines *numerischen Ausdrucks* zurück.

• Details

Der *numerische Ausdruck* muß größer als 0 sein.
e ist näherungsweise 2.718281828459046.
Alle logarithmischen Funktionen werden normalerweise in einfacher Genauigkeit berechnet. Wenn der *numerische Ausdruck* eine Zahl doppelter Genauigkeit ist, wird der Logarithmus ebenfalls doppelt genau berechnet.

• Siehe auch

`EXP, ^ Operator`

• Beispiel

```
PRINT LOG10(10),LOG10(5)
```

LPOS Funktion

• Syntax

`LPOS(`*Dummy*`)`

• Wirkungsweise

Diese Funktion gibt die Anzahl der Zeichen wieder, die seit dem letzten Carriage-Return/Line-Feed mit LPRINT gedruckt wurden.

• Details

Es wird nicht die aktuelle physikalische Position des Druckkopfs zurückgegeben, ebenso wie Tabulatoren nicht auf Leerstellen erweitert werden. Der Rückgabewert ist die Position im Druckerpuffer beginnend mit 1.
Dummy ist ein Parameter ohne Bedeutung.

• Siehe auch

`POS, LPRINT.`

• Beispiel

```
' Diese Routine druckt solange Strings, bis der
' Druckkopf die 50. Spalte erreicht hat.
WHILE LPOS(0)<50
   I%=I%+1
   LPRINT String$(I%)
WEND
```

LPRINT, LPRINT USING Befehle

• Syntax

```
LPRINT [Liste][{;|,}]
LPRINT USING Format_String; Liste [{;|,}]
```

• Wirkungsweise

Diese Befehle geben Daten auf dem aktuellen Drucker-Port aus.

• Details

LPRINT und LPRINT USING verhalten sich analog zu PRINT und PRINT USING, mit dem Unterschied, daß die Daten hier auf dem aktuellen Drucker (PRT:) ausgegeben werden. Den Drucker kann man mit dem Preferences-Tool des Betriebssystems auswählen.
Um eine bessere Kontrolle über den Drucker zu erhalten, kann man Kanal 256 in OPEN- und CLOSE-Befehlen benutzen. Normalerweise wird dieser Kanal direkt vor dem ersten LPRINT geöffnet und bei Programmende wieder geschlossen.
Wenn man Druckersteuerzeichen (für z.B. Fettschrift oder Sonderzeichen) senden will, sollte man lieber die hardwareunabhängigen Drucker-Escape-Sequenzen aus dem AMIGA-Handbuch benutzen, als die aus dem Handbuch zum Drucker. Wir empfehlen die AMIGA-Sequenzen deshalb, weil, wenn man einen anderen Drucker unterstützen will (oder jemanden sein Programm gibt, der nicht denselben Drucker hat), nur den neuen Druckertreiber installieren muß, und nicht sein Programm überarbeiten muß.
Wenn man spezielle Fähigkeiten seines Druckers ausnutzen will, die nicht über die AMIGA-Sequenzen zu realisieren sind, muß man Kanal 256 des gewünschten Gerätes (PAR: oder SER:) öffnen.

• Siehe auch

```
PRINT, LPRINT USING, PCOPY
```

• Beispiel

```
LPRINT CHR$(12); ' Sendet ein Form-Feed zum am
                 ' Drucker-Port angeschlossenen
                 ' Gerät.
```

LSET Befehl

• Syntax

```
LSET String_Variable = String_Ausdruck
```

• Wirkungsweise

Macht eine *String-Variable* linksbündig. Normalerweise wird dieser Befehl in Zusammenhang mit dem FIELD-Befehl gebraucht.

• Details

Der LSET-Befehl macht eine String-Variable linksbündig, indem er sie auf ihre volle Länge von rechts mit Leerstellen auffüllt. Normalerweise wird LSET in Zusammenhang mit dem FIELD-Befehl benutzt; man kann es aber auch zur formatierten Ausgabe von ganz gewöhnlichen String-Variablen nutzen. RSET ist ähnlich zu LSET, nur wird hier rechtsbündig formatiert.

• Siehe auch

```
FIELD, RSET, PUT
```

• Beispiel

```
FIELD #2,20 AS a$
LSET a$=HoleDaten$(10)
PUT #2
```

LTRIM$ Funktion

• Syntax

LTRIM$(*String_Ausdruck*)

• Wirkungsweise

Diese Funktion kappt führende Leerstellen eines *String-Ausdrucks*.

• Details

LTRIM$ kappt Leerstellen eines Strings von links aus. Ein Anwendungsgebiet wäre das Kürzen von Kommandozeileneingaben um führende Leerstellen.

• Siehe auch

RTRIM$

• Beispiele

```
Kommandozeile$=LTRIM$(KOMMANDO$)
' Kommandozeile$ soll keine führende Leerstellen
' haben.

PRINT LTRIM$("   Maxon")
PRINT LTRIM$("BASIC")
```

Ergebnis:

```
Maxon
BASIC
```

MAX Funktion

• Syntax

`MAX(Ausdruck,Ausdruck [,Ausdruck]...)`

• Wirkungsweise

Findet das Maximum zweier oder mehrerer Werte.

• Details

MAX findet den größten aus einer Anzahl von Ausdrücken. Die Ausdrükke können sowohl numerisch als auch Strings sein.

• Siehe auch

`MIN`

• Beispiel

```
INPUT "Geben Sie drei Namen ein: ";a$,b$,c$
PRINT "Der erste Name lautet: ";MIN(a$,b$,c$)
PRINT "Der letzte Name lautet: ";MAX(a$,b$,c$)
```

MENU Befehl

• Syntax

```
MENU Titel_ID,Eintrag_ID,Status[,Name_String]
MENU RESET
MENU {ON|OFF|STOP}
```

• Wirkungsweise

Dieser Befehl fügt den Text oder den Status eines Menüeintrags oder -titels hinzu oder verändert ihn. MENU RESET löscht alle hinzugefügten Einträge. Die MENU ON-, MENU OFF- und MENU STOP-Befehle dienen zur Kontrolle der Menüereignisse.

• Details

Titel_ID bestimmt, welches Menü verändert werden soll; dazu können Werte von 1 bis 10 verwendet werden. Der Wert 1 steht für das äußerst links stehende Menü.
Eintrag_ID bestimmt, welcher Eintrag des Menüs zu verändern ist, es sei denn sein Wert beträgt 0, denn dies gibt an, daß der Titel verändert werden soll.
Status bietet folgende Möglichkeiten:

0 Deaktivieren des Menüs oder Menütitels
1 Aktivieren des Menüs oder Menütitels
2 Aktivieren des Menüeintrags und Setzen eines Häkchens daneben.

Wenn der *Name_String$*-Parameter angegeben wird, wird dieser String als Name des Menüeintrags oder -titels genommen. Wenn man einen *Status* von 2 angibt, sollte man sicherstellen, daß der *Name_String$* für diesen Eintrag mit zwei Leerstellen beginnt.
Um herauszufinden, welcher Menüeintrag ausgewählt wurde, kann man entweder mit der MENU-Funktion (siehe unten) darauf warten, bis ein Menü angeklickt wurde, oder den ON MENU...GOSUB-Befehl benutzen.
Mit MENU ON kann die Überprüfung der Menüereignisse (Menu Event Checking) eingeschaltet werden. Das bewirkt, daß ON MENU-Befehle aktiviert werden, sobald auf den passenden Eintrag geklickt wurde.
Mit dem MENU OFF-Befehl erreicht man, daß Menüklicks (außer in dem Wert, der von der MENU-Funktion zurückgegeben wird) ignoriert werden.
MENU STOP bewirkt, daß Menüereignisse zwischengespeichert werden, bis ein MENU ON-Befehl ausgeführt wird. ON

MENU...GOSUB ist dann aktiviert. Diese Speicherung ist dann sinnvoll, wenn man zeitkritischen Code ausführen will, da die permanente Abfrage der Menüs einige Prozessorzeit kostet.

• Siehe auch

`MENU Funktion, ON MENU...GOSUB`

• Beispiel

```
MENU 1,0,1,"Projekt"
MENU 1,1,1,"Laden"
MENU 1,2,1,"Speichern"

MENU 2,0,1,"Suchen"
MENU 2,1,1, "  Finden"

MENU 1,2,0 ' schaltet den Speichern-Eintrag aus
MENU 2,1,2 ' setzt ein tick mark vor den Find-
             Eintrag
MENU 2,1,1 ' löscht das Häkchen wieder
```

MENU Funktion

• Syntax

`MENU(n)`

• Wirkungsweise

Diese Funktion gibt den Titel und den Eintrag des zuletzt ausgewählten Menüeintrags zurück.

• Details

MENU(0) gibt normalerweise eine 0 zurück, es sei denn ein Menüeintrag wurde seit dem letzten Aufruf der Funktion angewählt; in diesem Fall wird die Nummer des Menütitels (von 1 bis 10) zurückgegeben. Bei weiteren MENU(0)-Aufrufen wird anschließend wieder eine 0 zurückgegeben, solange bis wieder ein Eintrag angewählt wurde.
MENU(1) gibt die Nummer (von 1 bis 19) des zuletzt ausgewählten Menüeintrags zurück. MENU(2) gibt einen Untereintrag zurück, insofern ein Menü überhaupt einen hat.

• Siehe auch

`MENU Befehl, ON MENU...GOSUB`

• Beispiel

```
MENU 1,0,1,"Projekt"
MENU 1,1,1,"Laden"
MENU 1,2,1,"Speichern"
MENU 1,3,1,"Ende"

MENU 2,0,1,"Suchen"
MENU 2,1,1,"  Finden"

DO
   DO
        Titel%=menu(0)
   LOOP WHILE Titel%=0
   PRINT "Titel: ";Titel%;" Eintrag:;MENU(1)
LOOP UNTIL Titel%=1 AND MENU(1)=3    'Ende
```

MID$ Funktion

• Syntax

MID$(*String_Ausdruck*,*n*[,*Laenge*])

• Wirkungsweise

Dieser Befehl schneidet Zeichen aus einem *String-Ausdruck*. Dabei gibt *n* die Position des ersten Zeichens und *Laenge* die Anzahl der Zeichen, die ausgeschnitten werden sollen, an. MID$ erzeugt einen neuen String.

• Details

Wenn der *String-Ausdruck* weniger Zeichen enthält, als in *Laenge* angegeben, oder weniger als *n* Zeichen nach rechts enthält, werden alle dem *n*-ten Zeichen folgenden Zeichen zurückgegeben. Wenn weniger als *n* Zeichen in dem *String-Ausdruck* enthalten sind, wird ein Leerstring ("") zurückgegeben.

• Siehe auch

RIGHT$, LEFT$, SADD

• Beispiel

```
String$="Natürlich MaxonBASIC, was sonst?"
PRINT MID$(String$,11,5)
```

Ergebnis:

```
Maxon
```

MID$ Befehl

• Syntax

```
MID$(String_Variable,n[,Laenge])=String_Ausdruck
```

• Wirkungsweise

Ersetzt einen Teil einer String-Variablen.

• Details

Die Zeichen einer *String_Variablen,* beginnend ab Position *n,* werden in einem *String_Ausdruck* ersetzt. Wenn der *Laenge*-Parameter angegeben wurde, wird nur die dort genannte Anzahl an Zeichen ersetzt; anderenfalls wird der ganze *String_Ausdruck* benutzt.
MID$ kann nicht dazu angewandt werden, die Länge eines Strings zu verändern. Bei der *String_Variablen* kann es sich um eine einfache String-Variable oder ein Element eines String-Arrays handeln. Sowohl *n* als auch *Laenge* müssen Integer- oder Long-Integer-Zahlen sein.

• Siehe auch

```
LSET, RSET
```

• Beispiel

```
Satz$="Hier ist ein Beispiel"
MID$(Satz$,1)=" Das war"
PRINT Satz$
```

Ergebnis:

```
 Das war ein Beispiel
```

MIN Funktion

• Syntax

`MIN(Ausdruck,Ausdruck [,Ausdruck]...)`

• Wirkungsweise

Findet das Minimum zweier oder mehrerer Werte.

• Details

MIN findet den kleinsten aus einer Anzahl von Ausdrücken. Die Ausdrücke können sowohl numerisch als auch Strings sein.

• Siehe auch

`MAX`

• Beispiel

```
INPUT "Geben Sie drei Werte ein: ";a,b,c
PRINT "Der kleinste Wert lautet: ";MIN(a,b,c)
```

MKDIR Befehl

• Syntax

```
MKDIR Pfadname
```

• Wirkungsweise

Dieser Befehl erzeugt ein Directory-Unterverzeichnis, dessen Name durch *Pfadname* angegeben ist.

• Details

Der *Pfadname* muß den AmigaDOS-Richtlinien entsprechen.
MKDIR kann sowohl relativ als auch also absolut zum *Pfadnamen* benutzt werden. Nähere Informationen finden Sie unter dem CHDIR-Befehl.

• Siehe auch

CHDIR, CURDIR$, RMDIR.

• Beispiel

```
MKDIR "WORK:EINS" 'ein absoluter Pfadname
CHDIR "WORK:EINS"
MKDIR "ZWEI"      'ein relativer Pfadname;
                  'erzeugt WORK:EINS/ZWEI
```

MKI$, MKFFP$, MKL$, MKS$, MKD$ Funktionen

• Syntax

MKI$(*Integer_Ausdruck*)
MKFFP$(*einfach_genauer_Ausdruck*)
MKL$(*Long_Integer_Ausdruck*)
MKS$(*einfach_genauer_Ausdruck*)
MKD$(*doppelt_genauer_Ausdruck*)

• Wirkungsweise

Diese Funktionen wandeln numerische Daten in Strings um.

• Details

Diese Funktionen nehmen einen numerischen Wert und speichern ihn in einer String-Variablen. Das Ergebnis ist binär, also kein normal lesbarer ASCII-Code. Diese Funktionen werden oft vor der Ausgabe numerischer Daten mittels LSET und RSET benutzt; beide Befehle können nur mit Strings arbeiten. Man unbedingt darauf achten, daß diese Funktionen nicht mit dem STR$-Befehl austauschbar sind. Ein numerischer Wert, der mit einer MKtyp$-Funktion gewandelt und anschließend ausgedruckt wurde, stellt immer die Zahl in binärer Schreibweise dar. Obwohl MaxonBASIC zum Interpreter ein unterschiedliches Format für Variablen mit einfacher Genauigkeit benutzt, wird es von MKS$- und CVS-Funktionen verstanden; dabei kann allerdings ein Genauigkeitsverlaust entstehen. Um dies zu vermeiden, kann man die MKFFP$- und CVFFP-Funktionen benutzen; allerdings darf man sie nicht für Dateien, die in Zusammenhang mit dem Interpreter gelesen oder geschrieben werden sollen, verwenden.

• Siehe auch

CVI, CVD, CVS, CVL, FIX, INT, STR$.

• Beispiel

```
OPEN "FELD.DAT" AS #1
FIELD #1, 4 AS Lint$    'definiert ein Feld
Langint&=65536
LSET Lint$=MKL$(Langint&) 'legt den String-Wert
                          'im Record ab und
PUT #1,1                'schreibt ihn in die Datei
CLOSE
```

MOUSE Funktion

• Syntax

```
MOUSE(Attribut)
```

• Wirkungsweise

Liest die aktuelle Mausposition und den Status der linken Maustaste.

• Details

Es wird eine Integer-Zahl zurückgegeben, deren Interpretation abhängig vom *Attribut* ist:

0 Status der linken Maustaste:
- 0 Die linke Maustaste ist nicht gedrückt und wurde es auch nicht seit dem letzten MOUSE(0)-Aufruf.
- 1 Die linke Maustaste ist nicht gedrückt, es erfolgte allerdings ein Einfachklick seit dem letzten MOUSE(0)-Aufruf.
- 2 Die linke Maustaste ist gedrückt, es erfolgte außerdem ein Doppelklick seit dem letzten MOUSE(0)-Aufruf.
- -1 Der Anwender bewegt die Maus, die Maustaste ist gedrückt und wurde einmal gedrückt.
- -2 Der Anwender hält die linke Maustaste gedrückt und hat einen Doppelklick gemacht.

1 Aktuelle x-Position der Maus

2 Aktuelle y-Position der Maus

3 x-Position der Maus beim letzten Klick, bevor der MOUSE(0)-Aufruf erfolgte.

4 y-Position der Maus beim letzten Klick, bevor der MOUSE(0)-Aufruf erfolgte.

5 x-Position der Maus beim Loslassen der Taste, sofern die Taste schon losgelassen war, als der MOUSE(0)-Aufruf erfolgte; wenn die Taste gedrückt war, gibt MOUSE(5) die x-Position beim MOUSE(0)-Aufruf zurück.

6 y-Position der Maus beim Loslassen der Taste, insofern die Taste schon losgelassen war, als der MOUSE(0)-Aufruf erfolgte; wenn die Taste gedrückt war, gibt MOUSE(6) die y-Position beim MOUSE(0)-Aufruf zurück.

• Siehe auch

MOUSE Befehl, ON...MOUSE

• Beispiel

```
DO
   'zeichnet Punkte mit der linken Maustaste
   IF MOUSE(0)<>0 THEN PSET(MOUSE(1),MOUSE(2))
LOOP UNTIL INKEY$=CHR$(27)'bis Esc gedrückt wird
```

MOUSE Befehl

• Syntax

```
MOUSE {ON|OFF|STOP}
```

• Wirkungsweise

Ändert die Mausereignisüberwachung (Mouse Event Trapping), um unterschiedlich auf ON...MOUSE-Befehle reagieren zu können.

• Details

Mit MOUSE ON läßt sich die Mausereignisüberwachung einschalten. Das bewirkt, daß ON MOUSE-Befehle aktiv werden, sobald der Anwender mit der linken Maustaste klickt.
Nach einem MOUSE OFF-Befehl werden Menüklicks ignoriert (außer der Rückgabewert wird von der MOUSE-Funktion geliefert).
MOUSE STOP bewirkt, daß Mausereignisse zwischengespeichert werden, bis ein MOUSE ON-Befehl ausgeführt wird. Ab diesem Zeitpunkt werden ON MOUSE...GOSUB-Befehle wieder aktiv. Die Zwischenspeicherung ist bei zeitkritischen Programmteilen sinnvoll.

• Beispiel

```
MOUSE STOP    ' Mausereignisüberwachung ist aus

...
MOUSE ON      ' Mausereignisüberwachung ist an
```

NAME Befehl

• Syntax

```
NAME alter_Dateiname AS neuer_Dateiname
```

• Wirkungsweise

Dieser Befehl benennt eine bestimmte Datei um.

• Details

Beide Dateinamen, der alte wie der neue, müssen den AmigaDOS-Regeln entsprechen. Man kann auch Dateien zwischen verschiedenen Verzeichnissen eines Laufwerks verschieben, indem man einfach umbenennt. Beide Dateinamen sind Strings.

• Siehe auch

```
KILL.
```

• Beispiel

```
NAME "ALT.BAS" AS "NEU.BAS"
```

OBJECT Befehle & Funktionen

• Syntax

```
OBJECT.AX Objekt_ID,Wert
OBJECT.AY Objekt_ID,Wert
OBJECT.CLIP (x1,y1)-(x2,y2)
OBJECT.CLOSE [Objekt_ID[,Objekt_ID...]]
OBJECT.HIT Objekt_ID,[MeineMaske]
                                [,KollisionsMaske]
OBJECT.ON [Objekt_ID[,Objekt_ID...]]
OBJECT.OFF [Objekt_ID[,Objekt_ID...]]
OBJECT.PLANES
Objekt_ID[,Ebenenwahl][,Ebene_an_aus]
OBJECT.SHAPE Objekt_ID1,{String|Objekt_ID2}
OBJECT.START [Objekt_ID[,Objekt_ID...]]
OBJECT.STOP [Objekt_ID[,Objekt_ID...]]
OBJECT.VX Objekt_ID,Wert
OBJECT.VY Objekt_ID,Wert
OBJECT.VX(Objekt_ID)
OBJECT.VY(Objekt_ID)
OBJECT.X Objekt_ID,Wert
OBJECT.Y Objekt_ID,Wert
OBJECT.X(Objekt_ID)
OBJECT.Y(Objekt_ID)
```

• Wirkungsweise

Hierbei handelt es sich um die AmigaBASIC-kompatiblen Objekt-(oder Sprite-)Befehle und -Funktionen.

• Details

Objekte verfügen immer über eine *Objekt_ID*. Diese ID besteht aus einer Integer-Zahl zwischen 1 und 32767 und bezieht sich immer auf das aktuelle Fenster (z.B. können zwei verschiedene Fenster verschiedene Objekte mit der gleichen ID verwalten, obwohl das ziemlich verwirrend sein dürfte). Objekte müssen mit dem OBJECT.SHAPE-Befehl definiert werden, bevor irgendein anderer Objekt-Befehl benutzt werden kann.

OBJECT.AX Objekt_ID,Wert
Hiermit wird die horizontale Beschleunigung eines angegebenen Objekts mit dem *Wert* Pixel/Sek.^{2} bestimmt.

OBJECT.AY Objekt_ID,Wert
Hiermit wird die vertikale Beschleunigung eines angegebenen Objekts mit dem *Wert* Pixel/Sek.^{2} bestimmt.

OBJECT.CLIP (x1,y1)-(x2,y2)
Bestimmt das Rechteck, in dem Objekte für ein Fenster gezeichnet werden. Sobald der erste OBJECT.SHAPE-Befehl für ein Fenster aufgerufen wurde, ist die ganze Fläche dieses Fensters zunächst einmal als ein solches Rechteck definiert.

OBJECT.CLOSE [Objekt_ID[,Objekt_ID...]]
Löscht eines oder mehrere angegebene Objekte und gibt deren Speicherreservierung frei. Wenn keine *Objekt_ID* angegeben wird, werden alle Objekte des aktuellen Fensters gelöscht.

OBJECT.HIT Objekt_ID,[MeineMaske][,KollisionsMaske]
Läßt Objekte einander passieren. Das durch die *Objekt_ID* angegebene Objekt kollidiert nur mit anderen Objekten, deren *KollisionsMaske*, wenn sie mit *MeineMaske* verundet (mit AND logisch verknüpft) wird, ein Ergebnis ungleich 0 zurückgibt. Wenn keine Maske angegeben wird, bleibt alles so, wie es war.

OBJECT.ON [Objekt_ID[,Objekt_ID...]]
Zeigt die angegebenen Objekte auf dem Bildschirm. Wenn keine Objekte definiert wurden, werden alle Objekte des Fensters angezeigt.

OBJECT.OFF Objekt_ID[,Objekt_ID...]]
Stoppt die Anzeige der angegebenen Objekte, gibt allerdings nicht den von ihnen genutzten Speicher frei. Wenn kein Objekt angegeben wird, werden alle Objekte des Fensters versteckt.

OBJECT.PLANES Objekt_ID[,Ebenenwahl][,Ebene_an_aus]
Bestimmt die *Ebenenwahl-* und *Ebene_an_aus*-Masken eines angegebenen Objekts. Wenn kein Parameter angegeben wird, bleibt alles so, wie es war.

OBJECT.SHAPE Objekt_ID1,{String|Objekt_ID2}
Wenn dieser Befehl mit einem *String*-Ausdruck angewandt wird, definiert er ein neues Objekt; die einfachste Methode, solche Strings zu definieren, ist mittels des Programms ObjEdit.

Das Format sieht folgendermaßen aus:

1-8 unbenutzt, sollte 0 sein

9-12 Anzahl der Bit-Ebenen

13-16 Breite eines Objekts in Pixeln

17-20 Höhe eines Objekts in Pixeln

21-22 Flags bitweise
- 1 wenn VSprite
- 8 sichere Hintergrund, bevor BOB gezeichnet wird
- 16 Farbe 0 für BOB ist transparent, nicht Schwarz
- 32 Bob überschreibt Hintergrund

23-24 initialisiere *Ebenenwahl*-Wert

25-26 initialisiere *Ebenen_an_aus*-Wert

27-... Ebenendaten

Wenn der zweite Ausdruck eine Objekt_ID ist, benutzt das neue Objekt dieselben Bildschirmdaten wie ein bereits existierendes Objekt.

OBJECT.START [Objekt_ID[,Objekt_ID...]]
Hiermit wird ein angegebenes Objekt bewegt, wenn man für es mit OBJECT.VX/VY/AX/AY eine Geschwindigkeit oder Beschleunigung definiert hat. Wenn kein bestimmtes Objekt angegeben wurde, werden alle Objekte dieses Fensters gestartet.

OBJECT.STOP Objekt_ID[,Objekt_ID...]]
Hiermit werden die angegebenen Objekte angehalten. Wenn kein bestimmtes Objekt angegeben wurde, werden alle Objekte dieses Fensters angehalten.

OBJECT.VX Objekt_ID,Wert
Hiermit wird die horizontale Geschwindigkeit eines Objekts in Pixel/Sek. bestimmt.

OBJECT.VY Objekt_ID,Wert
Hiermit wird die vertikale Geschwindigkeit eines Objekts in Pixel/Sek. bestimmt.

OBJECT.VX(Objekt_ID)
Diese Funktion gibt die aktuelle horizontale Geschwindigkeit eines Objekts in Pixel/Sek. zurück.

OBJECT.VY(Objekt_ID)
Diese Funktion gibt die aktuelle vertikale Geschwindigkeit eines Objekts in Pixel/Sek. zurück.

OBJECT.X Objekt_ID,Wert
Hiermit wird die x-Koordinate eines bestimmten Objekts gesetzt.

OBJECT.Y Objekt_ID,Wert
Hiermit wird die y-Koordinate eines bestimmten Objekts gesetzt.

OBJECT.X(Objekt_ID)
Gibt die aktuelle x-Koordinate eines bestimmten Objekts zurück.

OBJECT.Y(Objekt_ID)
Gibt die aktuelle y-Koordinate eines bestimmten Objekts zurück.

OCT$ function

• Syntax

```
OCT$(numerischer_Ausdruck)
```

• Wirkungsweise

Diese Funktion gibt einen String zurück, der einen *numerischen_Ausdruck* in oktaler Schreibweise enthält.

• Details

OCT$ gibt den Integer-Teil eines *numerischen_Ausdrucks* zurück. Wenn der Ausdruck eine Integer-Zahl ist, ist der Ergebnis-String 1 bis 6 Zeichen lang, bei einer Long-Integer-Zahl kann er bis zu 11 Zeichen lang sein.

• Siehe auch

```
BIN$, HEX$
```

• Beispiel

```
PRINT OCT$(8)
```

Ergebnis:

```
10
```

ON...BREAK Befehl

• Syntax

```
ON BREAK GOSUB {Zeilennummer|Zeilenlabel|0}
```

• Wirkungsweise

Springt eine Unterroutine an, die aufgerufen wird, wenn die Ctrl-C- oder .-Taste gedrückt wird.

• Details

ON BREAK GOTO 0 veranlaßt MaxonBASIC, die Abbruchereignisse (Break Events) selbst zu bearbeiten.
Mit BREAK ON wird die Bearbeitung der Abbruchereignisse eingeschaltet.

• Siehe auch

```
BREAK
```

• Beispiel

```
ON BREAK GOSUB BreakHandler
BREAK ON
...
BreakHandler:
RETURN ' macht nichts und ignoriert folglich die
       'Abbruch-Tasten
```

ON...CLOSE Befehl

• Syntax

```
ON CLOSE GOSUB {Zeilennummer|Zeilenlabel|0}
```

• Wirkungsweise

Springt eine Unterroutine an, die aufgerufen wird, wenn der Anwender ein Fenster geschlossen hat.

• Details

ON CLOSE GOTO 0 bewirkt, daß MaxonBASIC die Fenster-Schließ-Ereignisse (Window Closure Events) selbst bearbeitet, indem das Fenster geschlossen wird, ohne das dem Programm zu melden. Wenn das letzte Fenster geschlossen ist, wird das Programm beendet.
Die ON...CLOSE-Unterroutine kann bestimmen, welche Fenster mit der WINDOW(9)-Funktion geschlossen werden. Wenn das Programm das Fenster schließen will, muß es dies selbst mit WINDOW CLOSE machen.
Das Fenster-Schließ-Ereignis-Handling ist normalerweise eingeschaltet, kann aber mit CLOSE OFF-, CLOSE ON- und CLOSE STOP-Befehlen verändert werden.

• Siehe auch

```
CLOSE,WINDOW Befehl und Funktion
```

• Beispiel

```
ON CLOSE GOSUB schliess_handler
...
schliess_handler:
   INPUT "Wollen Sie wirklich dieses Fenster
                                        schließen?";a$
   IF UCASE$(a$)="JA" THEN WINDOW CLOSE WINDOW(9)
   RETURN
```

ON...COLLISION Befehl

• Syntax

```
ON COLLISION GOSUB {Zeilennummer|Zeilenlabel|0}
```

• Wirkungsweise

Springt eine Unterroutine an, die aufgerufen wird, wenn ein Objekt mit dem Rand oder einem anderen Objekt kollidiert.

• Details

ON COLLISION GOTO 0 veranlaßt MaxonBASIC, Kollisionsereignisse (Collision Events) zu ignorieren.
Mit COLLISION ON kann man die Kollisionsereignisbearbeitung einschalten.

• Siehe auch

```
OBJECT,COLLISION
```

• Beispiel

```
ON COLLISION GOSUB KollisionsHandler
```

ON...ERROR Befehl

• Syntax

```
ON ERROR GOTO {Zeilennummer|Zeilenlabel}
```

• Wirkungsweise

Schaltet das Fehler-Handling ein und bestimmt die Fehler-Handling-Routine.

• Details

Dieser Befehl erlaubt, Laufzeitfehler, wie z.B. Überlauf oder Diskette-voll, abzufangen und die Kontrolle an eine bestimmte BASIC-Zeile zu übergeben, sobald ein Fehler auftritt.
Wenn eine Zeilennummer 0 verwendet wird, ist das Fehler-Handling ausgeschaltet, und Fehler haben zur Folge, daß das Programm ganz normal verlassen wird. Wenn ON ERROR GOTO 0 im Fehler-Handler verwendet wird, wird die original Fehlermeldung ausgegeben und das Programm angehalten.
Will man das Programm nach einem Fehler weiterlaufen lassen, muß man den RESUME-Befehl anwenden.
Fehler-Handler sollten mit Unterprogrammen und -funktionen realisiert werden – wenn das Unterprogramm oder die -funktion beendet sind, kehrt die Fehlerroutine dorthin zurück, von wo aus sie aufgerufen wurde. Diese MaxonBASIC-Erweiterung ermöglicht, daß man eigene Library-Unterprogrammen schreiben kann, die ihre eigene Fehlerbehandlung haben, ohne daß diese die Fehlerbehandlungsroutinen anderer Programme verstehen müssen.
Benutzt man ON ERROR in einem Programm, wird dadurch das Programm größer und etwas langsamer in der Ausführung. Dies kommt daher, daß sich das BASIC zusätzliche Informationen während der Programmausführung merken muß.

• Siehe auch

RESUME, ERL, ERR

• Beispiel

```
ON ERROR GOTO Handler
OPEN "DATEN.INF" FOR INPUT AS #2
INPUT LINE #2,a$
CLOSE #2
PRINT "Die Daten lauten ";a$ : STOP
Handler:
IF ERR=53 THEN
  PRINT "Fehler - Die Datei DATEN.INF wurde
                          nicht gefunden" : STOP
ELSE
  ON ERROR GOTO 0    'ein anderer Fehler
END IF
```

ON...GOSUB Befehl

• Syntax

```
ON n GOSUB {Zeilennummer|Zeilenlabel}
                    [,{Zeilennummer|Zeilenlabel...}]...
```

• Wirkungsweise

Dieser Befehl ruft die n-te aus einer Liste von Unterroutinen auf, je nach Wert des Parameters..

• Details

Hat *n* Nachkommastellen, wird *n* zu einer Integer-Zahl gerundet.
Auf welche Routine der Liste verzweigt wird, hängt vom Parameter *n* ab. Wenn *n* den Wert 1 hat, wird auf die erste Unterroutine verzweigt, bei 2 auf die zweite usw.
Ist *n* kleiner als 1 oder größer als die Anzahl der verfügbaren Unterroutinen, fährt das Programm direkt hinter dem ON...GOSUB-Befehl mit der Ausführung fort.
Es ist normalerweise besser einen SELECT CASE-Befehl und einen Unterprogrammaufruf zu verwenden als einen ON...GOSUB-Befehl.

• Siehe auch

SELECT...CASE, ON...GOTO

• Beispiel

```
ON Nummer% GOSUB Routine1, Routine2, Routine3
' Wenn Nummer% = 3, wird Routine3 ausgeführt.
```

ON...GOTO Befehl

• Syntax

```
ON n GOTO {Zeilennummer|Label}
                          [,{Zeilennummer|Label}]...
```

• Wirkungsweise

Dieser Befehl bewirkt, daß ein Programm zur n-ten einer Liste von Programmzeilen verzweigt.

• Details

Dieser Befehl unterscheidet sich von ON...GOSUB darin, daß ON...GOTO nicht in die Zeile nach diesem Befehl zurückkehrt.
Hat *n* Nachkommastellen, wird es zu einer Integer-Zahl gerundet.
n bestimmt, zu welchem Label verzweigt werden soll. Ist *n* größer als die Anzahl der angegebenen Labels oder kleiner als 1, fährt das Programm direkt hinter dem ON...GOTO-Befehl fort.
Es ist besser, den SELECT CASE-Befehl zu benutzen als ON...GOTO.

• Siehe auch

`SELECT...CASE, ON...GOSUB.`

• Beispiel

```
ON Nummer% GOTO Verzweigung1, Verzweigung2
'Ist Nummer% = 1, verzweigt das Programm zu
'Verzweigung1.
```

ON...MENU Befehl

• Syntax

```
ON MENU GOSUB {Zeilennummer|Zeilenlabel|0}
```

• Wirkungsweise

Bestimmt die Unterroutine, die angesprungen wird, wenn der Anwender auf einen Menüeintrag klickt.

• Details

ON MENU GOTO 0 schaltet die Menüereignisüberwachung (Menu Event) aus.
Mit MENU ON wird sie wieder eingeschaltet. Um herauszufinden, welcher Menüeintrag ausgewählt wurde, muß man die MENU-Funktion verwenden.

• Siehe auch

MENU Befehl, MENU Funktion

• Beispiel

```
MENU 1,0,1,"Projekt"
MENU 1,1,1,"Laden"
MENU 1,2,1,"Speichern"
MENU 1,3,1,"Ende"

MENU 2,0,1,"Suchen"
MENU 2,1,1,"  Finden"

ON MENU GOSUB MenueHandler
MENU ON  'schaltet die Überwachung der
         'Menüereignisse ein
' Hier steht das Programm
SLEEP    ' und wartet auf ein Ereignis

MenueHandler:
   titel%=menu(0)
   PRINT "Titel ";titel%;" Objekt: ";MENU(1)
   IF titel%=1 AND MENU(1)=3 THEN STOP   'Ende
   RETURN   'Ende
```

ON...MOUSE Befehl

• Syntax

```
ON MOUSE GOSUB {Zeilennummer|Zeilenlabel|0}
```

• Wirkungsweise

Bestimmt eine Unterroutine, die aufgerufen wird, wenn der Anwender auf die linke Maustaste drückt.

• Details

ON MOUSE GOTO 0 schaltet die Mausereignisüberwachung (Mouse Event) aus.
Mit MOUSE ON wird sie wieder eingeschaltet. Um die Mausposition herauszufinden, muß man die MOUSE-Funktion verwenden.

• Siehe auch

MOUSE Befehl, MOUSE Funktion

• Beispiel

```
ON MOUSE GOSUB MausHandler
'Hier steht das Hauptprogramm
DO
   SLEEP     ' wartet auf ein Ereignis
LOOP
MausHandler:
   ereignis%=MOUSE(0) 'hole nächstes Mausereignis
   PSET (MOUSE(1),MOUSE(1)
   RETURN
```

ô ô

ON...TIMER Befehl

• Syntax

```
ON TIMER(n) GOSUB {Zeilennummer|Zeilenlabel}
ON TIMER GOSUB 0
```

• Wirkungsweise

Setzt oder deaktiviert ein Timer-Ereignis (Timer Event).

• Details

n enthält die Frequenz des Timer-Ereignisses in Sekunden. Dabei handelt es sich um eine Fließkommzahl mit einfacher Genauigkeit, die zwischen 0 und 86400 liegen muß.
ON TIMER GOTO 0 schaltet das Timer-Ereignis aus.
Mit TIMER ON kann man die Timer-Ereignisüberwachung aktivieren.

• Siehe auch

`TIMER.`

• Beispiel

```
ON TIMER(5) GOSUB TimerHandler
'Hier steht das Hauptprogramm
DO
   SLEEP
LOOP
   'wartet auf Ereignisse

TimerHandler:
   PRINT "Timer-Ereignisse finden alle 5 Sekunden
                                              statt."
   RETURN    'Ende
```

OPEN Befehl

• Syntax

```
OPEN Pfadname [FOR modus] AS [#]Kanalnummer
                              [LEN=Record_Groesse]
```

oder

```
OPEN Modus_String, [#]Kanalnummer, Pfadname
                              [,Record_Groesse]
```

• Wirkungsweise

Dieser Befehl öffnet einen Datenkanal zum Lesen oder Schreiben einer Datei.

• Details

Pfadname muß den AmigaDOS-Spezifikationen entsprechen.
modus kann einer der folgenden Modi sein:

APPEND
Gibt eine sequentielle Datei an, an die Daten angehängt werden sollen.

INPUT
Gibt eine sequentielle Datei an, von der Daten gelesen werden sollen.

OUTPUT
Gibt eine sequentielle Datei an, in die Daten geschrieben werden sollen.

RANDOM
Gibt eine Datei mit wahlfreiem Zugriff an, in die Daten geschrieben werden sollen oder von der gelesen werden soll.

Der *Modus_String* ist ein String-Ausdruck, der aus einem Buchstaben besteht, der dem ersten Buchstaben des *Modus* der Datei (A, I, O or R) – in Groß- oder Kleinschrift – entspricht.
Kanalnummer kann eine Integer-Zahl von 1 bis 255 inklusiv sein.
Record_Groesse bestimmt die Länge des Records einer Datei mit wahlfreiem Zugriff in Bytes oder die Größe des internen Puffers für andere Dateitypen. Der voreingestellte Wert für *Record_Groesse* ist 128 Bytes.
OPEN bezieht sich auf eine Datei oder ein Gerät mit *Kanalnummer*. Diese Nummer wird bei allen Lese- und Schreiboperationen benutzt, um auf Dateien oder Geräte zugreifen zu können.

Das Gegenstück zu OPEN bildet CLOSE. Es empfiehlt sich, sooft wie möglich eine Datei mit CLOSE zu schließen und später bei Bedarf wieder zu öffnen. Das dient zur eigenen Sicherheit; wenn ihr Computer aus irgendwelchen Gründen abstürzt, sind alle Daten, die sich zu diesem Zeitpunkt im Datenpuffer befinden, verloren, und die Datei kann unter Umständen nur noch unlesbare Daten enthalten.
Normalerweise bezieht sich der *Pfadname* auf eine Datei. Stattdessen kann aber auch einer von bereits festgelegten Gerätenamen angegeben werden. Diese Namen lauten:

PAR:	paralleler Drucker-Port (Centronics)
SER:	serieller RS232-Port
PRT:	aktueller Drucker-Port, wie mit Preferences definiert
LPT1:	wie PRT:
COMx:	serieller Port. Siehe weiter unten für weitere Informationen.

Kanäle, die mit diesen Namen geöffnet werden, können nur mit einfachen Ein-/Ausgabebefehlen, wie PRINT#, WRITE#, INPUT# und INPUT$ benutzt werden. Operationen mit Dateien mit wahlfreiem Zugriff und andere Befehle (wie z.B. LOF), die keinen Sinn bei Nicht-Laufwerken ergeben, funktionieren nicht. Die Ausnahme hierzu findet sich in der LOC-Funktion, die auch mit COMx funktioniert, wenn man diesen Port anspricht, um die Anzahl von Zeichen zu erhalten, die derzeit verfügbar sind. Wenn man zuvor mit HiSoftBASIC 1 programmiert hat, muß man darauf achten, daß dieses BASIC niemals mehr als 1 Zeichen als verfügbar meldet.
COMx: ermöglicht eine erweiterte Kontrolle über die seriellen Schnittstellen. Die Syntax sieht folgendermaßen aus:

```
COM[Port_Nummer],[Baud_Rate],[Parity],[Datenbits],
        [Stoppbits][,Extra_Flags[,Extra_Flags...]]
```

Die Parameter haben folgende Funktionen:

Port_Nummer
Diese Nummer bestimmt, welche serielle Schnittstelle benutzt werden soll. 1 ist die interne serielle Schnittstelle (Gerät 0) und 2 bis 9 ist für weitere serielle Schnittstellen (Geräte 1 bis 8) reserviert. Die Nutzung weiterer serieller Schnittstellen ist ein besonderes MaxonBASIC-Feature.

Baud_Rate

Dieser Parameter bestimmt die Übertragungsgeschwindigkeit der Zeichen bei der Ein- und Ausgabe über serielle Schnittstellen. Der AMIGA-Dokumentation entsprechend soll die Übertragungsrate zwischen 112 und 19200 Baud liegen. Höhere Baud-Raten können zu Datenverlust bei der Übertragung führen. Voreingestellt ist 300 Baud.

Parity
Bestimmt die Parität (Prüf-Bit wird gesetzt und kontrolliert) bei der Datenübertragung. Folgende Möglichkeiten sind einstellbar:

E even (gerade; ist voreingestellt)
O odd (ungerade)
N none (kein Prüf-Bit)
F fast (schnell). Hiermit wird das RAD_BOOGIE-Bit gesetzt, das die Überprüfung der Parität, der Abbruchprüfung und den XON/XOFF-Handshake ausschaltet.

Datenbits
5 bis 8 Bits möglich, voreingestellt 7 Bits

Stoppbits
0 bis 3 Bits möglich, voreingestellt 1 Bit

Extra_Flags
Folgende *Extra-Flags* sind spezielle MaxonBASIC-Erweiterungen; es können mehrere davon in beliebiger Reihenfolge gleichzeitig benutzt werden:

H schaltet den Hardware-Handshake ein (setzt das 7WIRE-Bit)
X schaltet den Software-Handshake (XON/XOFF) aus (setzt das XDISABLED-Bit)
Bnnn bestimmt die Größe des Puffers für seriellen Geräte. Die Größe wird in dezimalen Bytes angegeben. Das Verändern der Puffergröße löscht eventuell noch im Puffer befindliche Daten. Die voreingestellte Größe des Puffers beträgt 512 Bytes und läßt sich auch über die Serial Preferences einstellen.

Wenn man OPEN für einen Datenkanal benutzt, der bereits geöffnet ist, wird die alte Datei zuerst geschlossen, bevor die neue geöffnet wird. Wenn die Datei dabei für einen wahlfreien Zugriff (Random Access) geöffnet ist, werden alle Variablen, die mit FIELD definiert wurden, gelöscht.

• Siehe auch

CLOSE.

• Beispiel

```
OPEN "RECORDS.DAT" FOR RANDOM AS #1 LEN=32
'Diese Zeile öffnet die Datei RECORDS.DAT für
'einen wahlfreien Lese-/Schreibzugriff,
'Kanalnummer 1, Record-Länge 32 Bytes.
'Die nächste Zeile hat fast dieselbe Funktion,
'benutzt aber eine andere Syntax und öffnet
'Datenkanal 2, nicht 1.

OPEN "R",#2,"RECORDS.DAT",32

OPEN "COM1:9600,N,8,1,H,X" FOR OUTPUT AS 1
'öffnet die interne serielle Schnittstelle zur
'Ausgabe mit 9600 Baud, 8 Datenbits, keiner
'Parität, mit Hardware- und ohne Software-
'Handshake.

OPEN "COM1:9600,N,8,1,B32768" AS 1
'öffnet die interne serielle Schnittstelle zur
'Ein- und Ausgabe mit 9600 Baud, 8 Datenbits,
'keiner Parität und Puffergröße von 32768 Bytes.
```

OPTION BASE Befehl

• Syntax

```
OPTION BASE {0|1}
```

• Wirkungsweise

Dieser Befehl definiert den kleinsten Index eines Arrays.

• Details

OPTION BASE bestimmt den Indexwert des ersten Elements in einem Array. Z.B. bewirkt OPTION BASE 1, daß mit DIM Array%(42) eine Anzahl von 42 Elementen (1-42) alloziert wird (bei der Grundeinstellung von OPTION BASE, nämlich 0, wären es 43 (0-42) gewesen. Wenn man OPTION BASE 1 benutzt, spart das kaum Speicherplatz, es sei denn, man benutzt drei- oder höherdimensionale Arrays.
Achtung: Wenn die Array-Überprüfoption (Array Check Option) ausgeschaltet ist und das Programm compiliert wird, hat OPTION BASE keine Auswirkung, da automatisch die Basis 0 angenommen wird.

• Siehe auch

```
DIM
```

• Beispiel

```
OPTION BASE 0  'Grundeinstellung
DIM Anderes_Array%(177)  ' 178 Elements werden
                         ' alloziert, 0 bis 177
```

PAINT Befehl

• Syntax

```
PAINT [STEP] (x,y) [,Zeichenfarbe] [,Randfarbe]
```

• Wirkungsweise

Füllt eine geschlossene Fläche mit einer bestimmten Farbe.

• Details

Die Fläche, die gefüllt werden soll, basiert auf *(x,y)* und muß mit Pixeln in der *Randfarbe* umschlossen sein.
Die Füllfarbe wird mit *Zeichenfarbe* bestimmt. Wird keine Farbe angegeben, wird die Fläche mit der aktuellen Vordergrundfarbe gefüllt; diese kann mit dem COLOR-Befehl geändert werden. Farbschattierungen lassen sich mit dem PALETTE-Befehl ändern.
Wird die *Randfarbe* weggelassen, wird statt dessen die *Zeichenfarbe* genommen.
PAINT läßt die aktuelle Grafikposition an dem angegeben Punkt.
Dieser Befehl kann nur in Zusammenhang mit Fenstern des Typs 16 bis 31 (z.B. Smart oder Super-Bitmap) benutzt werden.

• Siehe auch

CIRCLE

• Beispiele

```
CIRCLE (100,100),50,3,2
PAINT (100,100),3,2
' zeichnet einen gefüllten Kreis mit der
' Zeichenfarbe 3 und Randfarbe 2

CIRCLE (100,100),50
PAINT STEP (0,0)
' zeichnet denselben Kreis in der aktuellen
' Vordergrundfarbe
```

PALETTE Befehl

• Syntax

PALETTE ***Farbnummer,Rot,Gruen,Blau***

• Wirkungsweise

Dieser Befehl erlaubt, die physikalische Erscheinungsweise von Farben auf dem Bildschirm zu verändern.

• Details

Jede Farbe auf dem Amiga kann ihre eigene aktuelle Erscheinungsweise haben, d.h. jeder Farbnummer kann eine andere Farbe zugeordnet werden. Die Erscheinungsweise kann auf dem Bildschirm mittels der Farbpalette geändert werden. Die Anzahl der möglichen Farben hängt von der Anzahl der Bit-Ebenen des aktuellen Bildschirms ab und kann von 0 bis zu 255 liegen. Allerdings sind in der Grundeinstellung nur die Farben 0 bis 3 nutzbar.
Die Parameter *Rot*, *Gruen* und *Blau* geben das Mischungsverhältnis der Grundfarben für die gewünschte *Farbnummer* an. Dieses Mischungsverhältnis kann von 0 zu 1 variieren, wobei 0 „diese Grundfarbe nicht", also 0%, und 1 „diese Grundfarbe voll", also 100%, bedeutet. Sofern die Hardware entsprechend ausgelegt ist, kann man mit diesem Befehl Farben aus einer Palette von bis zu 24 Bit Farbauflösung bestimmen. *Farbnummern* sind auch als *Farb_IDs* bekannt.

• Beispiel

```
PALETTE 1,0,0,0 'definiert Farbe 1 als Schwarz
PALETTE 0,1,1,1 'definiert Farbe 0 als Weiß

PALETTE 2,1,0,0 'definiert Farbe 2 als
                'Grundfarbe Rot
```

PATTERN Befehl

• Syntax

`PATTERN [Linienmuster][,Flächenfüllmuster_Array]`

• Wirkungsweise

Definiert eine Linienmuster und/oder ein Flächenfüllmuster

• Details

Bei *Linienmuster* handelt es sich um einen Integer-Ausdruck, der dazu dient eine Bit-Maske für ein Linienmuster zu definieren. Normalerweise empfiehlt es sich, dies in hexadezimaler oder binärer Schreibweise anzugeben. Siehe auch das Beispiel unten.
Flächenfüllmuster_Array ist ein Integer-Array, das dem *Linienmuster* ähnelt; es wird in Zusammenhang mit dem AREAFILL-Befehl benutzt. Jeder Eintrag in dem Array wird als 16 Bit horizontale Maske gewertet. Das komplette Array wird verwendet; es sollte sich um ein eindimensionales Array handeln, das aus einer Anzahl von Elementen, die auf einer 2er-Potenz (z.B. 2, 4, 8, 16 etc.) basiert, besteht. Es sind keine Klammern nach dem Array-Namen erforderlich.

• Beispiel

```
PATTERN &h5555 ' zeichnet fein gepunktete Linien
PATTERN &b0101010101010101
                ' dasselbe in Binärdarstellung

PATTERN &9999   ' gröber gepunktete Linien

DIM PAT%(3)
PAT%(0)=&h8888
PAT%(1)=&h4444
PAT%(2)=&h2222
PAT%(3)=&h1111

PATTERN ,PAT%   ' ein sehr feines graues
                ' Füllmuster
```

PCOPY Befehl

• Syntax

```
PCOPY
```

• Wirkungsweise

Gibt den aktuellen Bildschirm auf einem Drucker aus.

• Details

Dieser Befehl bewirkt dasselbe, als wenn man das ScreenPrint-Programm von der Amiga Extras-Diskette benutzt. Er benutzt die Druckerkonfiguration, die mit dem Preferences-Programm definiert wurde.

• Siehe auch

```
LPRINT
```

• Beispiel

```
IF a$="C" THEN
   MOUSE -1 ' verstecke Maus
   PCOPY    ' und drucke Bildschirm
END IF
```

PEEK, PEEKB, PEEKL, PEEKW, PEEK$ Funktionen

• Syntax

```
PEEK(Adresse)
PEEKB(Adresse)
PEEKL(gerade_Adresse)
PEEKW(gerade_Adresse)
PEEK$(Adresse)
```

• Wirkungsweise

Diese Funktionen geben den Inhalt einer bestimmten Speicherstelle zurück.

• Details

PEEKB gibt ein Byte zurück, PEEKW ein Wort (16 Bit) und PEEKL ein Langwort (32 Bit). PEEK$ liest einen – wie in C – nullterminierten String an der *Adresse* aus und konvertiert ihn in einen BASIC-String.
Wenn man PEEKL oder PEEKW verwendet, muß die angegebene Adresse gerade sein, anderenfalls erhält man einen Fehler (Fatal Error – Unexpected Exception Error), es sei denn, man hat einen Amiga mit mindestens einem 68020-Prozessor.
Mit diesen Funktionen kann man am schnellsten den Anwenderspeicherbereich auslesen. Aus diesem Grund finden keine weiteren Überprüfungen statt; lediglich bei PEEKL und PEEKW wird geprüft, ob die angegebene Adresse gerade ist.
PEEK, PEEKB und PEEKW geben Integer-Werte zurück, PEEKL einen Long-Integer-Wert und PEEK$ einen String.
In MaxonBASIC entspricht die PEEK- genau der PEEKB-Funktion. Um kompatibel zum AmigaBASIC-Interpreter zu sein, muß man PEEK verwenden.
Diese Funktionen werden oft dazu genutzt, um Werte auszulesen, die von Betriebssystem-Libraries zurückgegeben werden.

• Siehe auch

`POKE, POKEB, POKEW, POKEL`

• Beispiel

```
Sieben%=7
PRINT PEEKW(VARPTR(Sieben%))
```

Ergebnis:
7

```
' Dieses Beispiel benötigt Workbench 2
DEFINT a-z
REM $include asl.bh
LIBRARY OPEN "asl.library"
CONST TAG_DONE&=0
DIM frtags&(20)
 TAGLIST VARPTR(frtags&(0)), _ASL_TitleText&,
                    "MaxonBASIC Datei Requester",
                                        _TAG_DONE&

 fr&=AllocAslRequest&(ASL_FileRequest&,
VARPTR(frtags&(0)))
 IF fr& THEN
   IF AslRequest&(fr&,0) THEN
      PRINT "Datei "+PEEK$(PEEKL(fr&+fr_File))
      PRINT "Dir "+PEEK$(PEEKL(fr&+fr_Drawer))
   ELSE
        PRINT "Abbruch"
   END IF
   FreeASlRequest fr&
 END IF
```

POINT Funktion

• Syntax

```
POINT(x_Pixel,y_Pixel)
```

• Wirkungsweise

Diese Funktion gibt die Farbe eines bestimmten Pixels zurück.

• Details

x_Pixel und *y_Pixel* geben die Koordinaten des Pixels, dessen Farbe abgefragt werden soll, im aktuellen Fenster an. Das Ergebnis entspricht der Tabelle, die man bei der Beschreibung des COLOR-Befehls findet. Befindet sich das gesuchte Pixel außerhalb des Fensterbereichs, wird als Ergebnis eine -1 zurückgegeben.

• Siehe auch

```
PSET, PRESET, LINE
```

• Beispiel

```
PRINT "B"
FOR x=0 to 8
FOR y=0 to 16
IF POINT(x,y) THEN
        LINE (10*(x+1),10*(y+2))-STEP(10,10),,bf
NEXT y,x
```

POKE, POKEB, POKEL, POKEW Befehle

• Syntax

```
POKE Adresse, Byte_Wert
POKEB Adresse, Byte_Wert
POKEL gerade_Adresse, Langwort_Wert
POKEW gerade_Adresse, Wort_Wert
```

• Wirkungsweise

Diese Befehle schreiben Daten direkt in den Arbeitsspeicher.

• Details

Diese Befehle schreiben entweder ein Byte, ein Wort oder Langwort an *Adresse*.
Schreibt man mit POKEL oder POKEW auf einem Amiga mit 68000-Prozessor an eine ungerade Adresse, erhält man eine Fehlermeldung (Fatal Error – Address Exception).
In MaxonBASIC entspricht die POKE- genau der POKEB-Funktion. Um kompatibel zum AmigaBASIC-Interpreter zu sein, muß man POKE verwenden.

• Beispiel

```
Zehn%=10
POKEW VARPTR(Zehn%),7
PRINT Zehn%
```

Ergebnis:

```
 7
```

POS Funktion

• Syntax

```
POS(x)
```

• Wirkungsweise

Diese Funktion gibt die Spalte der aktuellen Cursor-Position zurück.

• Details

x ist ein Argument ohne Funktion. Der Wert, der zurückgegeben wird, ist die Spaltenposition (horizontale Position) des Cursors. Steht der Cursor äußerst links, ist der Rückgabewert 1. *x* muß numerisch sein.

• Siehe auch

```
LPOS, LOCATE, PRINT
```

• Beispiel

```
x%=POS(0)
PRINT "Der Cursor befindet sich in Spalte ";x%
```

PRESET Befehl

• Syntax

```
PRESET [STEP] (x_Pos,y_Pos) [,Farbe]
```

• Wirkungsweise

Dieser Befehl löscht oder setzt ein Pixel in der angegebenen Farbe im aktuellen Fenster.

• Details

PRESET verhält sich ähnlich wie PSET, mit der Ausnahme, daß die Hintergrundfarbe benutzt wird, wenn der *Farbe*-Parameter weggelassen wird. Die Parameter *x_Pos* und *y_Pos* geben den Punkt an, an dem das Pixel gesetzt werden soll, relativ zur oberen linken Ecke des aktuellen Fensters. Wenn der optionale STEP-Parameter benutzt wird, werden die Koordinaten relativ zur aktuellen Grafikposition angesehen.

• Siehe auch

PSET, POINT, LINE

• Beispiel

```
CONST ypos=70
DEFINT i
FOR i=10 TO 350
   IF i<300 THEN PSET (i,ypos)
   IF i>50 THEN PRESET (i-50,ypos)
NEXT i
```

PRINT Befehl

• Syntax

`PRINT [Ausdruck1][{;| |,}Ausdruck2][;]...`

• Wirkungsweise

Dieser Befehl gibt Daten auf dem Bildschirm aus, die in einer Liste mit Ausdrücken angegeben werden müssen.

• Details

Die Liste mit Ausrücken kann aus Strings und/oder numerischen Ausdrücken bestehen. Diese müssen mit Semikolons, Kommata oder Leerstellen getrennt sein. Endet eine Liste mit Ausdrücken nicht mit einem Semikolon oder Komma, wird ein Line-Feed ausgegeben.
Wird PRINT ohne Parameter oder Trennzeichen aufgerufen, wird ebenfalls ein Line-Feed ausgegeben, folglich in die nächste Zeile gegangen.
Wird ein Semikolon benutzt, werden die Ausdrücke ohne Leerstellen zwischen ihnen ausgegeben. Eine Leerstelle zwischen den Parametern hat denselben Effekt wie ein Semikolon. Wir empfehlen aufgrund der besseren Übersichtlichkeit, ein Semikolon zu verwenden. Die Angabe eines Kommas bewirkt, daß der Ausdruck nach dem Komma an der nächsten Tabulatorposition ausgegeben wird.
SPC, TAB und PTAB dürfen in PRINT-Befehlen benutzt werden, sonst aber nirgendwo.
Nummern werden entweder mit einer vorangestellten Leerstelle (positiv) oder einem Minuszeichen (negativ) und immer mit einer nachfolgenden Leerstelle ausgegeben.
Ein Fragezeichen kann für PRINT verwendet werden; dabei handelt es sich um ein Feature älterer BASIC-Interpreter, das aus Kompatibiltätsgründen mit in MaxonBASIC aufgenommen wurde.
Man muß beachten, daß aus Kompatibilitätsgründen zum AmigaBASIC die voreingestellte Breite eines Fensters unbegrenzt ist. Man sollte also folgende Zeilen nicht benutzen,

```
FOR I=1 TO 100
   PRINT I,
NEXT I
```

ohne daß die Breite des Fensters zuvor mit WIDTH neu festgelegt wurde; anderenfalls erscheinen alle 100 Nummern in derselben Zeile.

• See

LPRINT, PRINT #, PRINT USING, WRITE

• Beispiel

```
PRINT "Der Bereich geht von ";1;" bis ";100,"
                                        (inklusive)"
```

Ergebnis:

```
Der Bereich geht von 1 bis 100 (inklusive)
```

PRINT#, PRINT# USING Befehle

• Syntax

```
PRINT #Kanalnummer, [USING Format_String]
                                Ausdrucksliste [;]
```

• Wirkungsweise

Diese Befehle schreiben formatierte Daten zu einer Datei oder auf ein Gerät.

• Details

Kanalnummer ist die Nummer, die angegeben werden muß, wenn die Datei geöffnet wird.
Format_String ist ein optionaler String mit formatierten Zeichen. Nähere Informationen hierzu finden Sie bei PRINT USING.
Ausdrucksliste ist die Liste von String und/oder numerischen Ausdrükken, die geschrieben werden sollen.
PRINT# schreibt die Daten in genau derselben Art und Weise, wie dies mit PRINT auf den Bildschirm geschieht. Aus diesemGrund muß man sehr genau darauf achten, welches Format man schreibt, damit die Daten bei einem späteren INPUT# wieder eingelesen werden können.

• Siehe auch

PUT, WRITE, INPUT#, FORMATx$

• Beispiel

```
FOR I%=1 TO 5
   PRINT #1,I%;",";
NEXT I%
```

Ergebnis: (in der Zieldatei)

```
 1 , 2 , 3 , 4 , 5 ,
```

PRINT USING Befehl

• Syntax

PRINT USING ***Format_String; Ausdrucksliste*** **[{,|;}]**

• Wirkungsweise

Dieser Befehl gibt eine *Ausdrucksliste* aus, die entsprechend dem *Format_String* formatiert ist.

• Details

Die *Ausdrucksliste* enthält die Ausdrücke (numerisch oder String), die ausgegeben werden sollen; sie müssen durch Semikolons voneinander getrennt sein.
Format_String ist ein String von Formatzeichen, die die Form der Ausgabe bestimmen.
Es gibt drei Formatzeichen, um einen String zu formatieren:

!	nur das erste Zeichen des Strings wird ausgegeben
\\	*2 + n* Zeichen des Strings werden ausgegeben. Bei *n* handelt es sich um die Anzahl der Leerstellen zwischen den Backslashes (\). Ist der String länger als *2 + n*, werden die übrigen Zeichen nicht ausgegeben. Ist *2 + n* größer als die Anzahl der Zeichen in dem String, wird der String linksbündig und mit Leerzeichen nach rechts aufgefüllt ausgegeben.
&	Bestimmt ein variables Längenfeld; besteht der *Format_String* aus einem Et-Zeichen (&) , wird der String ohne eine Veränderung ausgegeben.

Es gibt mehrere Formatzeichen, um einen numerischen Ausdruck zu formatieren:

#	Das Doppelkreuz (#) steht für jeweils eine Zahlenstelle, die ausgegeben werden soll. Hat eine Zahl weniger Stellen, als angegeben, wird sie rechtsbündig und mit Leerzeichen nach links aufgefüllt ausgegeben.
.	Ein Dezimalpunkt wird ausgegeben. Wird im *Format_String* bestimmt, daß sich eine Stelle vor dem Dezimalpunkt befindet, wird diese Stelle immer ausgegeben, auch wenn sie 0 lautet.
+	Bewirkt, daß das Vorzeichen der Zahl (+/–) vor oder hinter dem String ausgegeben wird, je nachdem, ob + vor oder hinter dem *Format_String* steht.
–	Steht ein Minuszeichen (–) am Ende des *Format_Strings*, wird ein Minuszeichen ausgegeben, wenn die Zahl negativ ist. Ist die Zahl positiv, hat das Minus keine Auswirkung.

** Bewirkt, daß die vorangestellten Leerzeichen einer Zahl mit Sternchen (Asterix) ersetzt werden; hat man also ein doppeltes Sternchen, bedeutet dies zwei Stellen mehr vor der Zahl.

$$ Das doppelte Dollar-Zeichen bewirkt, daß ein Dollar-Zeichen direkt vor die Zahl geschrieben wird. Dazu werden zwei Stellen mehr benötigt, wovon eine für das Dollar-Zeichen ist.

**$ Dies ist eine Kombination aus ** und $$; vorangestellte Leerzeichen sind Sternchen und ein $ wird vor der Zahl ausgegeben. Dazu werden drei Stellen mehr benötigt, wovon eine für das Dollar-Zeichen ist. Ist eine Zahl negativ, wird ein Minuszeichen direkt vor dem Dollar-Zeichen ausgegeben.

, Wird ein Komma vor einem Dezimalpunkt angegeben, wird links vom Dezimalpunkt alle drei Stellen ein Komma ausgegeben. Wird das Komma am Ende des *Format_Strings* angegeben, wird es als Teil der Nummer ausgegeben. Das Komma steht für eine zusätzliche Stelle und hat in Zusammmenarbeit mit ^^^^ keine Wirkung.

^^^^ Diese vier Akzente bestimmen die Darstellung eines Exponential-Formats. Die Leerstelle ist für die Ausgabe von E+xx reserviert. Der Dezimalpunkt kann an einer beliebigen Stelle in der Zahl stehen. Die Mantisse der Zahl ist linksbündig. Insofern ein vorangestelltes oder folgendes Plus (+) oder ein folgendes Minus (–) angegeben sind, wird eine Stelle links vom Dezimalpunkt das Vorzeichen der Zahl ausgegeben. Man muß darauf achten, daß Zahlen mit doppelter Genauigkeit fünf Exponenten haben können; man muß dann auch fünf Akzente (^) benutzen.

_ Ein Unterstrich vor einem Zeichen im *Format_String* bewirkt, daß das Zeichen als Buchstabe ausgegeben wird; ein Unterstrich wird ausgegeben, wenn man zwei aufeinanderfolgende Unterstriche im *Format_String* angibt.

Ein Prozentzeichen wird dann ausgegeben, wenn die Zahl, die ausgegeben werden soll, größer ist, als sie der *Format_String* darstellen kann. Ist eine gerundete Zahl größer als das Formatfeld, wird ein Prozentzeichen vor der Zahl angegeben.
Man kann das Währungssymbol, das ausgegeben wird, wenn man das Dollar-Zeichen ($) angibt, wechseln, indem man das Byte an der Adresse SYSTAB+38 ändert. Z.B. ändert folgende Zeile das Dollar- in ein Pfund-Zeichen:

```
POKEB SYSTAB+38,ASC("£")
```

• Siehe auch

PRINT, WRITE, FORMATx$.

• Beispiel

```
PRINT USING "!";"ALFRED"
PRINT USING "\ \";"JOHANNES"
PRINT USING "&";"Heute ist der"
PRINT USING "###.#";25.4
PRINT USING "#.#";2.54
PRINT USING "##.##  ";12.34,5.678,1.2
PRINT USING "+#.# ";-1.1,2.2
PRINT USING "#.#- ";-3.3,4.4
PRINT USING "**#.# ";56.78,-91.2,-3
PRINT USING "$$##.#";123.45
PRINT USING "**$##.##";2.54
PRINT USING "####,.##";4567.89
PRINT USING "+####^^^^";-0.4444
PRINT USING "_£##.##";4.99
PRINT USING "#.#";33.25
PRINT USING ".##";.999
```

Ergebnisse:

```
A
JOH
Heute ist der
 25.4
2.5
12.34   5.68   1.20
-1.1 +2.2
3.3- 4.4
*56.8 -91.2 *-3.0
$123.5
***$2.54
4,567.89
-4444E-04
£ 4.99
%33.3
.99
```

PSET Befehl

• Syntax

```
PSET [STEP] (x_Pos,y_Pos) [,Farbe]
```

• Wirkungsweise

Dieser Befehl zeichnet ein Pixel in einer angegebenen Farbe im aktuellen Fenster.

• Details

Der *Farbe*-Parameter gibt an, in welcher Farbe das Pixel gezeichnet werden soll; wird er weggelassen, wird mit der aktuellen Linienfarbe gezeichnet. Die Parameter *x_Pos* und *y_Pos* bestimmen die Position des Pixels relativ zur oberen linken Ecke des aktuellen Fensters. Wird das optionale STEP verwendet, werden die Koordinaten relativ zu den letzten PSET- oder PRESET-Koordinaten genommen.

• Siehe auch

PRESET, POINT, LINE.

• Beispiel

```
pi=3.1415926
FOR i=0 to 4*pi STEP 0.05
  PSET (i*50,100+100*SIN(i))
NEXT i
```

PTAB Funktion

• Syntax

```
PTAB(n)
```

• Wirkungsweise

Geht während der Ausgabe zu einer bestimmten Pixel-Position.

• Details

PTAB ähnelt der TAB-Funktion, mit der Ausnahme, daß hier die Position in Pixeln und nicht in Zeichen angegeben wird.
Befindet sich die aktuelle Ausgabeposition bereits hinter *n*, wird zu Position *n* in der nächsten Zeile gegangen. Ist *n* größer als die mögliche Breite für eine Ausgabe, wird zu der Position *n MOD Pixel-Breite* gegangen. Ist *n* kleiner als 1, wird die Ausgabeposition 1. PTAB darf nur in PRINT- und LPRINT-Befehlen benutzt werden.

• Siehe auch

```
PRINT, TAB.
```

• Beispiel

```
FOR i%=2 TO 200
   PRINT PTAB(i%);i%;
NEXT i%
```

PUT Ein-/Ausgabebefehl

• Syntax

```
PUT [#]Kanalnummer[,Record_Nummer]
```

• Wirkungsweise

Dieser Befehl schreibt einen Record aus dem Puffer für wahlfreien Zugriff (Random Access) in die angegebene Datei, die zuvor für wahlfreien Zugriff geöffnet wurde.

• Details

Kanalnummer ist die Nummer, die angegeben wurde, als die Datei geöffnet wurde.
Wird keine *Record_Nummer* angegeben, wird der geschriebene Record der nächste Record.

• Siehe auch

```
GET, OPEN, CLOSE
```

• Beispiel

```
LSET a$=Adresse$(10)
PUT #1,10
LSET a$=Adresse$(11)
PUT #1 'wird automatisch Record 11, da keine
       'Record-Nummer angegeben wurde.
```

PUT Grafikbefehl

• Syntax

```
PUT (x,y),Array_Name[(Arrayindex...)][,Modus]
```

• Wirkungsweise

Dieser Befehl kopiert (oder *blittet*) ein rechteckiges Bild, das mit GET gespeichert wurde, auf den Bildschirm.

• Details

Die obere linke Ecke des Rechtecks, relativ zum Bildschirm, wird durch *x* und *y* definiert. Wurde das Array nicht mittels eines GET-Befehls gefüllt, kann alles mögliche Undefinierbare angezeigt werden.
Mit dem optionalen *Modus*-Parameter kann der Kopiermodus bestimmt werden. Das kann in Form verschiedener reservierter Modi oder über eine Integer-Zahl geschehen. Erlaubte Modi sind:

PSET	kopiert das Bild direkt auf den Bildschirm und überschreibt somit den bisherigen Inhalt.
PRESET	ähnlich wie PSET, liefert allerdings ein Negativ des Bildes als Ergebnis.
AND, OR, XOR	das Bild wird logisch mit dem bisherigen Bildschirm verknüpft.

Der voreingestellte Modus ist XOR. Der Modus kann auch als Integer-Term beschrieben werden. Näheres darüber kann man in der Graphics Library des ROM Kernel Manuals unter *minterm* nachsehen.
Die Höhe und Breite des Bildes, das benutzt werden soll, ist durch das Bild bestimmt, das mit GET gespeichert wurde.
Es können mehrdimensionale Arrays in diesem Befehl benutzt werden, wodurch man leicht zwischen verschiedenen Bildern hin- und herschalten kann. Achtung: mehrdimensionale Arrays werden in entgegengesetzter Reihenfolge zu der Speichermethode des Interpreters gespeichert, so daß man AmigaBASIC-Programme entsprechend modifizieren muß.

• Siehe auch

GET.

• Beispiel

```
DIM scr%(4347-1) ' -1, da das Array mit 0 startet
GET (10,20)-(100,200),scr%
FOR i%=0 to 200 STEP 20
   PUT (i%,i%),scr%,PSET
NEXT i%
```

RANDOMIZE Befehl

• Syntax

RANDOMIZE [*Ausdruck*]

• Wirkungsweise

Dieser Befehl übergibt dem Zufallszahlengenerator eine neue Ausgangszahl.

• Details

Wird kein *Ausdruck* angegeben, fordert RANDOMIZE den Anwender zu einer Eingabe auf. *Ausdruck* muß ein Integer-Ausdruck sein. Wird TIMER als *Ausdruck* übergeben, liefert die TIMER-Funktion einen Ausgangswert.
Die RND-Funktion produziert immer dieselbe Sequenz an Zufallszahlen, bis eine neue Ausgangszahl übergeben wird.

• Siehe auch

RND

• Beispiel

```
Ausgangszahl%=42
RANDOMIZE Ausgangszahl%
Rand!=RND
```

READ Befehl

• Syntax

```
READ variable_list
```

• Wirkungsweise

Dieser Befehl weist Variablen Werte von DATA-Befehlen zu.

• Details

READ kann nur in Zusammenhang mit dem DATA-Befehl benutzt werden. Der Variablentyp des READ-Befehls wird in den Variablentyp des DATA-Befehls konvertiert.
Die Anzahl der Elemente eines DATA-Befehls sollte nicht die Anzahl der Variablen des READ-Befehls überschreiten. Sollten sich mehr Elemente in dem DATA-Befehl befinden, werden die zusätzlichen ignoriert.
Um die Elemente eines DATA-Befehls erneut einzulesen, benutzt man den RESTORE-Befehl.

• Siehe auch

```
DATA, INPUT$, INPUT
```

• Beispiel

```
DIM Array&(5)
FOR I%=1 TO 5
  READ Array&(I%)
NEXT I%
DATA &H42424242,&HFC0000,1200,-45002
```

REDIM Befehl

• Syntax

```
REDIM [PRESERVE|APPEND] Array(Indizes)
                                [,Array(Indizes)]...
```

• Wirkungsweise

Dieser Befehl ändert die Größe eines dynamischen Arrays.

• Details

Indizes können Integer- oder Long-Integer-Ausdrücke sein. Man kann REDIM auch dazu benutzen, die Anzahl der Dimensionen in einem Array zu ändern. Alle Daten, die sich in dem Array befinden, wenn es mit REDIM geändert wird, werden gelöscht.
Arrays, die mit REDIM verändert werden sollen, müssen dynamisch sein. Dazu müssen sie mit DIM mit einem Ausdruck (nicht einer Konstante) definiert werden oder REM $DYNAMIC muß bei der Definition aktiviert sein.
REDIM PRESERVE wird dazu benutzt, Speicher zu einem eindimensionalen Array hinzuzufügen, ohne die im Array enthaltene Information zu zerstören. Wenn der Index eines REDIM PRESERVE-Befehls kleiner ist als das original Array, wird das Array entsprechend gekappt; dabei gehen die übrig gebliebenen Daten nicht verloren.
REDIM APPEND entspricht REDIM PRESERVE und dient zur Kompatibilität zu HiSoft BASIC 1. REDIM PRESERVE sollte aus Kompatibilitätsgründen zu Microsoft BASIC 7.1 auf dem PC bevorzugt werden.

• Siehe auch

DIM, ERASE

• Beispiel

```
DIM Alfred%(256)
FOR I%=1 TO 256
   Alfred%(I%)=I%
NEXT I%
REDIM PRESERVE Alfred%(260)
PRINT Alfred%(42)
```

Result:

```
 42
```

REM Befehl

• Syntax

`REM` *`Kommentare ...`*

• Wirkungsweise

Dieser Befehl erlaubt Kommentare im Quelltext zu plazieren und definiert auch Meta-Befehle.

• Details

Kommentare können auch mit einem Apostroph „' " beginnen.
Folgt dem REM ein Dollar-Zeichen ($), wird es zum Meta-Befehl .
Anderenfalls wird der Rest der Zeile ignoriert und somit nicht als Befehl gewertet.

• Beispiel

```
' $ARRAY
REM Die vorangegangene Zeile schaltet die Array-
REM Überprüfung an.
```

REPEAT...END REPEAT Befehl

• Syntax

```
REPEAT Name
.
.
[EXIT Name]
.
.
END REPEAT Name
```

• Wirkungsweise

Die Befehle in einer REPEAT-Schleife werden solange ausgeführt, bis ein EXIT-Befehl für die Schleife ausgeführt wird.

• Details

Name ist ein gültiger Variablenname, der derzeit nicht im Programm benutzt wird. Man kann denselben Namen für eine Reihe verschiedener REPEAT-Schleifen mehrfach benutzen, allerdings darf es nicht der Name eines Unterprogramms, einer Variablen oder Funktion sein.
Diese Schleife ist die üblichste Schleifenkonstruktion in MaxonBASIC, da jede Schleife einen Namen bekommen kann. Außerdem kann man mit EXIT auch von einer äußeren in eine innere Schleife wechseln (siehe auch das Beispiel unten). Sind diese Möglichkeiten jedoch nicht gefordert, ist DO...LOOP normalerweise übersichtlicher, erfordert weniger Tipparbeit und ist zudem auch noch kompatibel zu Microsoft BASIC.
Achtung! REPEAT ist im EXIT-Befehl *nicht* definiert.

• Siehe auch

DO...LOOP, WHILE...WEND, FOR...NEXT.

• Beispiel

```
i=0: Summe=0
REPEAT Eins
   i=i+1: j=0
   REPEAT Zwei
      j=j+1
      Summe=Summe+a(i,j)
      IF Summe>10000 THEN
         PRINT "Viel zu groß": EXIT Eins
      END IF
      IF j=n THEN EXIT Zwei
   END REPEAT Zwei
   IF i=n THEN EXIT Eins
END REPEAT Eins
' Dieses Beispiel summiert die Elemente des
' zweidimensionalen Arrays a() und hält sofort
' an, wenn die Summe größer als 10000 wird.
```

RESET Befehl

• Syntax

```
RESET
```

• Wirkungsweise

Dieser Befehl schließt alle offenen Laufwerksdateien.

• Details

Alle Laufwerksschreibpuffer werden gespeichert und die Dateien geschlossen. Da alle Dateien automatisch bei Programmende geschlossen werden, wird dieser Befehl in der Regel nicht benötigt. Er dient zur Kompatiblität zu anderen BASIC-Dialekten.

• Siehe auch

```
CLEAR
```

• Beispiel

```
IF a$="QUIT" THEN RESET : STOP-1
' RESET ist hier eigentlich überflüssig
```

RESTORE Befehl

• Syntax

```
RESTORE [{Zeilennummer|Label}]
```

• Wirkungsweise

Dieser Befehl veranlaßt einen READ-Befehl auf einen bestimmten DATA-Befehl zuzugreifen, der zuvor eingelesen wurde.

• Details

Zeilennummer oder *Label* geben einen DATA-Befehl an, der noch einmal mit READ gelesen werden soll. Wird keine *Zeilennummer* oder ein *Label* angegeben, liest der nächste READ-Befehl den ersten DATA-Befehl, der in dem Programm vorkommt.

• Siehe auch

```
READ, DATA, INPUT
```

• Beispiel

```
READ Johannes&
PRINT Johannes&
Johannes&=0
PRINT Johannes&
RESTORE J_Daten
READ Johannes&
PRINT Johannes&
.
.
J_Daten:
DATA 3448325377,4020987683
```

Ergebnis:

```
 3448325377
 0
 3448325377
```

RESUME Befehl

• Syntax

```
RESUME {Zeilennummer|Zeilenlabel}
```

• Wirkungsweise

Dieser Befehl sorgt dafür, daß die Programmausführung nach einer Fehlerbehandlungsroutine in einer bestimmten Programmzeile fortfährt.

• Details

Dieser Befehl dient dazu, von einer Fehlerbehandlungsroutine, die mit ON ERROR GOTO aufgerufen wurde, in eine bestimmte Programmzeile zu gehen und dort mit dem Programm fortzufahren. Wird der Befehl außerhalb einer Fehlerbehandlungsroutine benutzt, erhält man einen Fehler (Fatal Error).
Es liegt an jedem Programmierer selbst, an die richtige Programmstelle zurückzukehren, an der der Fehler aufgetaucht ist. Um eine einwandfreie Programmausführung zu gewährleisten, muß man, wenn der Fehler in einem Unterprogramm, einer Funktion oder Unterroutine aufgetreten ist, auch dorthin zurückkehren. Anderenfalls können völlig unvorhersehbare Abstürze auftreten.

• Siehe auch

```
ON ERROR, ERL, ERROR
```

• Beispiel

```
   a$="TEST.DAT"
Neuer_Open_Versuch:
   ON ERROR GOTO kein_Open
   OPEN a$ FOR INPUT AS #1
   ON ERROR GOTO 0 ' Fehler werden gemeldet
   lese_Daten 1
   ......
kein_Open:
   PRINT "Datei nicht gefunden";a$
   INPUT "Dateiname:";a$
   RESUME Neuer_Open_Versuch
```

RETURN Befehl

• Syntax

```
RETURN [{Zeilennummer|Label}]
```

• Wirkungsweise

Dieser Befehl gibt die Programmausführung von einer Unterroutine an das Hauptprogramm zurück.

• Details

Die Programmausführung wird an die Zeile zurückgegeben, die hinter dem GOSUB-Befehl steht, der die Unterroutine aufgerufen hat. Wird eine bestimmte *Zeilennummer* oder *Label* angegeben, wird das Programm an dieser Stelle fortgeführt.
Achtung! Dieser Befehl ist nicht zum Verlassen eines Unterprogramms geeignet, man muß dafür EXIT SUB benutzen.

• Siehe auch

GOSUB, SUB, END, EXIT

• Beispiel

```
GOSUB Routine   ' verzweige zur Unterroutine
.               ' die Programmausführung wird
.               ' hier nach dem RETURN
.               ' fortgesetzt.
Routine:
.
.
RETURN          ' Rückkehr aus der Unterroutine
```

RIGHT$ Funktion

• Syntax

`RIGHT$(String_Ausdruck,n)`

• Wirkungsweise

Diese Funktion gibt einen Teilstring zurück, der *n* Zeichen des *String_Ausdrucks* von rechts beinhaltet.

• Details

Ist $n = 0$, wird ein Leerstring zurückgegeben. Ist *n* größer als der *String_Ausdruck*, wird der vollständige *String_Ausdruck* zurückgegeben.

• Siehe auch

`LEFT$, MID$, LEN`

• Beispiel

```
d$=DATE$
PRINT "Jahr:";RIGHT$(d$,4)
```

Ergebnis (im Jahr 1994):

```
Jahr: 1994
```

RINSTR Funktion

• Syntax

`RINSTR([Start,]String1,String2)`

• Wirkungsweise

Diese Funktion sucht rückwärts, ob *String2* in *String1* vorhanden ist.

• Details

Bei *Start* handelt es sich um einen optionalen Offset für den Beginn der Suche in *String1*. *Start* kann entweder eine Integer- oder eine Long-Integer-Zahl sein. Den höchsten Wert, den diese Funktion zurückgeben kann, ist *Start*. Strings können entweder String-Variablen, -Buchstaben oder -Ausdrücke sein. Der Rückgabewert ist die Position des ersten Zeichens des Strings, der gefunden wurde.
Wird *String2* in *String1* gefunden, wird die Stelle, an der er gefunden wurde, zurückgegeben.
Ist der Wert von *Start* größer als die Länge von *String1*, wird eine 0 zurückgegeben.
Ist *String1* ein Leerstring, wird eine 0 zurückgegeben.
Kann *String2* nicht gefunden werden, wird eine 0 zurückgegeben.
Ist *String2* ein Leerstring, wird *Start* zurückgegeben; wurde *Start* nicht definiert, wird eine 1 zurückgegeben.
Achtung! Da Strings in Maxon BASIC keine Längenbegrenzung haben, gibt diese Funktion eine Long-Integer-Zahl zurück.

• Siehe auch

`INSTR`

• Beispiel

```
' Diese Routine durchsucht die Kommandozeile
' nach einer Leerstelle, um den letzten
' Parameter zu finden, der an das Programm
' übergeben wurde.
I%=INSTR(KOMMANDO$," ")
IF I%=0 THEN I%=LEN(KOMMANDO$)
PRINT "Der letzte übergebene Parameter ist: ";
                              LEFT$(KOMMANDO$,I%)
```

RMDIR Befehl

• Syntax

```
RMDIR Pfadname
```

• Wirkungsweise

Dieser Befehl löscht ein leeres Unterverzeichnis eines Laufwerks.

• Details

Das Unterverzeichnis, das gelöscht werden soll, darf keine Dateien enthalten. Der *Pfadname* muß den Regeln des AmigaDOS entsprechen.

• Siehe auch

```
KILL, MKDIR, CHDIR.
```

• Beispiel

```
RMDIR "WORK:MUELL"
```

RND Funktion

• Syntax

```
RND[(n)]
```

• Wirkungsweise

Diese Funktion gibt eine Pseudo-Zufallszahl einfacher Genauigkeit zwischen 0 und 1 zurück.

• Details

Ist *n* = 0, wird die letzte erzeugte Nummer zurückgegeben. Ist *n* kleiner 0, wird *n* dazu benutzt, die Zahlensequenz zurückzusetzen. Ist *n* größer 0 oder wird weggelassen, wird die nächste Nummer aus der Sequenz erzeugt.
Der benutzte Algorithmus basiert auf dem von B.A. Wichman und I.D. Hill (NPL Report DITC 6/82).

• Siehe auch

`RANDOMIZE`

• Beispiel

```
Number!=RND(-1) ' Die Zahl, die zurückgegeben
                ' wird, hat -1 als Ursprung
```

RSET Befehl

• Syntax

```
RSET String_Variable = String_Ausdruck
```

• Wirkungsweise

Dieser Befehl schiebt Daten in den Puffer für Dateien mit wahlfreiem Zugriff (Random Access). Er kann auch dazu benutzt werden, einen Wert eines Strings in der *String_Variablen* rechtsbündig zu machen.

• Details

RSET bildet das Gegenstück zu LSET. Hat der *String_Ausdruck* weniger Bytes als die *String_Variable*, wird der String rechtsbündig mit Leerstellen aufgefüllt.

• Siehe auch

```
LSET, GET, FIELD
```

• Beispiel

```
z$=SPACE$(80)
RSET z$="Auf der rechten Seite des Bildschirms"
PRINT z$
```

RTRIM$ Funktion

• Syntax

`RTRIM$(String_Ausdruck)`

• Wirkungsweise

Diese Funktion gibt seinen String-Parameter zurück, nachdem alle folgenden Leerstellen abgeschnitten wurden.

• Details

RTRIM$ kappt Leerstellen von der rechten Seite aus. Es kann dazu benutzt werden, Leerstellen am Ende einer Eingabe abzuschneiden.

• Siehe auch

`LTRIM$, RIGHT$`

• Beispiele

```
Kommandozeile$=RTRIM$(LTRIM$(COMMAND$))
' Kommandozeile$ soll keine führenden oder
' folgenden Leerstellen haben.

PRINT RTRIM$("   Maxon   ");RTRIM$("BASIC")
```

Ergebnis:

```
MaxonBASIC
```

RUN Befehl

• Syntax

```
RUN [{Zeilennummer|Pfadname}]
```

• Wirkungsweise

Dieser Befehl startet das aktuelle Programm erneut oder das Programm, das durch den *Pfadnamen* angegeben ist.

• Details

Wenn eine *Zeilennummer* angegeben ist, startet das Programm ab dieser Zeile erneut. Alphanumerische Labels sind nicht erlaubt.
Pfadname kann ein gültiger String-Ausdruck sein und vom Typ her jede Art von ausführbarem Programm bezeichnen. In manchen BASIC-Dialekten sind die Anführungszeichen nicht notwendig, in MaxonBASIC müssen sie allerdings unbedingt angegeben werden.
Diese Möglichkeit sollte nicht benutzt werden, wenn man direkt in den Speicher compiliert.
Damit das funktioniert, muß das aufrufende Programm vom CLI oder dem MaxonBASIC-Editor aus gestartet werden.
RUN schließt alle offenen Dateien und löscht den aktuellen Arbeitsspeicher, bevor es das Programm startet.
Achtung! Dieser Befehl setzt voraus, daß das Programm entweder vom CLI oder via IconX gestartet wurde.

• Siehe auch

```
CHAIN
```

• Beispiel

```
RUN "MaxonASM"   ' startet MaxonASM im aktuellen
                 ' Verzeichnis
```

SADD Funktion

• Syntax

`SADD(String_Ausdruck)`

• Wirkungsweise

Diese Funktion gibt die Adresse eines *String_Ausdrucks* zurück.

• Details

Man muß mit dieser Funktion sehr vorsichtig umgehen, da Strings im Speicher ohne vorherige Warnung verschoben werden können, wenn eine Garbage Collection stattfindet. Deshalb sollte man die Adresse, die der SADD-Befehl liefert, direkt verwenden und nicht einer Variablen zuweisen.

• Siehe auch

`VARPTR`

• Beispiel

```
Astr$=" BASIC"
PRINT Astr$
POKEB SADD(Astr$),ASC("M")
PRINT Astr$
```

Ergebnis:

```
 BASIC
MBASIC
```

SAY Befehl

• Syntax

```
SAY String_Ausdruck[,Modus_Array]
```

• Wirkungsweise

Dieser Befehl läßt den Amiga unter älteren Betriebssystemversionen einen String von Phonemen „sprechen".

• Details

Dieser Befehl nutzt das **Narrator Device**, das ab Workbench 2.1 nicht mehr im Betriebssystem enthalten ist. Entwickler unterstützen diese Funktion nicht mehr, so daß wir ihre Nutzung nicht empfehlen können. Besitzern eines neueren Amigas (z.B. A1200 und A4000) fehlt dieses **Device**.

Modus_Array ist optional. Dieses Array und geeignete Phoneme sind ausführlich im AmigaBASIC-Handbuch beschrieben, so daß wir sie an dieser Stelle nicht weiter ausführen möchten.

Oft wird die TRANSLATE$-Funktion in Verbindung mit dem SAY-Befehl benutzt, um nicht Phoneme direkt konstruieren zu müssen.

Achtung! Unsere Standard-Disketten-Installation kopiert *nicht* die ***translator.library***, das ***narrator.device*** und den ***speak-handler*** auf die Boot-Diskette, die sie erzeugt; man muß diese Dateien selbst auf die Diskette kopieren und den MonAm-Debugger von ihr löschen.

• Siehe auch

```
TRANSLATE$
```

• Beispiel

```
SAY TRANSLATE$("Hello")
```

SCREEN Befehl

• Syntax

```
SCREEN Bildschirm_ID,Breite,Hoehe,Ebenen,
                                    Modus[,Typ]
```

oder

```
SCREEN CLOSE Bildschirm_ID
```

• Wirkungsweise

Dieser Befehl erzeugt oder schließt einen Amiga-Bildschirm.

• Details

Bildschirm_ID muß einen Wert zwischen 1 und 4 haben und bestimmt, welcher Bildschirm gemeint ist. Diese *Bildschirm_ID* kann in nachfolgenden WINDOW-Befehlen verwendet werden.
SCREEN CLOSE schließt den angegebenen Bildschirm und jedes Fenster, das noch auf diesem Bildschirm geöffnet ist.
Breite und *Hoehe* bestimmen die Größe des Bildschirms in Pixeln.
Ebenen legt die Anzahl der Bit-Ebenen des Bildschirms fest, so daß 1 Ebene 2 Farben, 2 Ebenen 4 Farben, 3 Ebenen 8 Farben, 4 Ebenen 16 Farben und 5 Ebenen 32 Farben ergibt.
Hat der *Modus*-Parameter den Wert 5, wird der Bildschirm im OS2-Look (3D-Knöpfe etc.) gezeichnet; dies geht aber erst ab Workbench 2 und höher. *Typ* ist die Display-ID des Bildschirms. Wird dieser weggelassen, wird der voreingestellte Wert der Workbench genommen.
Hat der *Modus*-Parameter nicht den Wert 5, handelt es sich um eine Definition in der alten Darstellungsart, die wie folgt definiert wird:

Wird der *Typ*-Parameter weggelassen, hat der *Modus*-Parameter folgende Bedeutungen:

1 niedrige Auflösung, non-interlaced
2 hohe Auflösung, non-interlaced
3 niedrige Auflösung, interlaced
4 hohe Auflösung, interlaced
5 frei definierbarer Bildschirm (Custom Screen)

Ist der *Typ*-Parameter angegeben, wird der *Modus*-Parameter ignoriert und der *Typ*-Parameter enthält den **ViewMode** des Bildschirms. Dabei handelt es sich um ein etwas komplexeres Thema, das ausführlich im ROM-Kernel-Handbuch beschrieben ist. Ein Bildschirm im HAM(Hold and Modify)-Modus in niedriger Auflösung kann mit einem *Typ* mit dem Wert &h800 definiert werden.

Achtung! Es findet keinerlei Überprüfung der Parameter des SCREEN-Befehls statt; das hat den Vorteil, daß man Grafikkarten o.ä. benutzen kann, hat aber auch den Nachteil, daß unerwartete Effekte auftreten können, wenn man die falsche Nummer angibt.
Dieser Befehl kann – je nach Auflösung – sehr leicht sehr viel Speicherplatz benötigen.

• Siehe auch

`WINDOW Befehl und Funktion`

• Beispiel

```
SCREEN 1,640,200,3,2   ' Bildschirm mit 8 Farben
                       ' in hoher Auflösung
SCREEN 1,640,200,8,5   ' 256 Farben, AGA
                       ' wenigstens erforderlich
```

SCROLL Befehl

• Syntax

```
SCROLL (x1,y1)-(x2-y2),delta_x,delta_y
```

• Wirkungsweise

Scrollt einen Ausschnitt des aktuellen Ausgabefensters.

• Details

(*x1*,*y1*)-(*x2*-*y2*) gibt das Rechteck an, das gescrollt werden soll. *delta_x* und *delta_y* bestimmt die Anzahl der Pixel, die nach rechts und nach unten gescrollt werden sollen. Negative Werte bewirken, daß nach links und nach oben gescrollt wird.

• Siehe auch

GET, PUT Grafikbefehle

• Beispiel

```
SCROLL (100,100)-(150,150),8,0

' scrollt eine Fläche von 50 mal 50 Pixeln um 8
' Pixel nach rechts.
```

SELECT...END SELECT Befehl

• Syntax

```
SELECT [CASE|ON] Variable
   [CASE|=] Case_Liste
        [EXIT SELECT]
.
.
   [CASE|=] Case_Liste
.
.
   [CASE|=] [ELSE|REMAINDER]
.
.
END SELECT
```

• Wirkungsweise

Eine Reihe von Befehlen kann – je nach Wert der *Variablen* – ausgeführt werden.

• Details

Bei der *Variablen* kann es sich um einen beliebige nichtindizierte Variable, egal ob numerisch oder String, handeln. Will man einen komplizierteren Ausdruck mit SELECT auswählen, sollte man diesen zuerst einer temporären Variablen zuweisen.
Die *Case_Listen* können folgende Inhalte, getrennt durch Kommata haben:

Ausdruck
oder
Ausdruck1 TO *Ausdruck2*
oder
Bedingungszeichen Ausdruck
oder
IS *Bedingungszeichen Ausdruck*

Wird der erste benutzt, werden, sofern die *Variable* gleich dem *Ausdruck* ist, die folgenden Befehle ausgeführt, bis der nächste CASE- oder =-Befehl auftaucht.
Wird die zweite Form benutzt, werden die folgenden Befehle ausgeführt, sofern die *Variable* größer oder gleich dem *Ausdruck1* und kleiner gleich dem *Ausdruck2* ist.

In der dritten und vierten Form kann das *Bedingungszeichen* eins der folgenden Zeichen sein: =,<>,>,>=,<=,< oder ==. Die darauf folgenden Befehle werden ausgeführt, wenn die *Variable* die Bedingung erfüllt. Ist eine CASE ELSE-, CASE REMAINDER-, =REMAINDER- oder =ELSE-Bedingung vorhanden, und keiner der CASE-Befehle trifft zu, werden die folgenden Befehle ausgeführt. Die CASE ELSE-Bedingung muß als letzte vor dem END SELECT stehen.
Die verschiedenen Alternativen sind zur leichteren Konvertierung von Programmen, die in anderen BASIC-Dialekten geschrieben wurden, vorgesehen. Microsoft BASIC benutzt SELECT CASE...CASE... CASE ELSE...END SELECT.

• Siehe auch

IF...THEN...ELSE

• Beispiele

```
INPUT A,B
SELECT CASE A
CASE 1,IS <0
   PRINT "negativ oder 1"
CASE IS >B
   PRINT "größer als B"
CASE 13 TO 19
   PRINT "in den Zehnern"
CASE ELSE
   PRINT "etwas anderes"
END SELECT
INPUT a$
SELECT a$
   ="JOHANNES","ALFRED": PRINT "Johannes oder
                                        Alfred"
   = <"A",>"Z": PRINT "Namen beginnen mit einem
                                  Großbuchstaben"
   = "MARIA": PRINT "Maria"
   = REMAINDER
        PRINT "unbekannter Name"
END SELECT
```

SGN Funktion

• Syntax

SGN(*numerischer_Ausdruck*)

• Wirkungsweise

Dieser Befehl gibt das Vorzeichen des *numerischen_Ausdrucks* zurück.

• Details

Ist der *numerische_Ausdruck* kleiner 0, wird –1 zurückgegeben.
Ist der *numerische_Ausdruck* gleich 0, wird 0 zurückgegeben.
Ist der *numerische_Ausdruck* größer 0, wird 1 zurückgegeben.
Der *numerische_Ausdruck* kann von beliebiger numerischer Art sein.

• Siehe auch

ABS

• Beispiel

```
Variable1% = -42
Variable2% = 0
Variable3% = 42
PRINT SGN(Variable1%)
PRINT SGN(Variable2%)
PRINT SGN(Variable3%)
```

Ergebnis:

```
-1
0
1
```

SHARED Befehl

• Syntax

`SHARED` *`Variablenliste`*

• Wirkungsweise

Dieser Befehl erlaubt einem Unterprogramm auf Variablen eines Hauptprogramms zuzugreifen, ohne daß sie zuvor als Parameter übergeben wurden.

• Details

Die *Variablenliste* kann Variablen- und Array-Namen enthalten, die mit () enden müssen.

• Siehe auch

`DIM, STATIC, LOCAL, COMMON SHARED`

• Beispiel

```
MAIN:
Alfred%=127
Johannes%=76
CALL Routine(Johannes%,10)
.
.
SUB Routine (A%,B%)
SHARED Alfred%
   Print Alfred%,A%,B%
END SUB
```

Ergebnis:

```
 127    76   10
```

SIN Funktion

• Syntax

`SIN(numerischer_Ausdruck)`

• Wirkungsweise

Diese Funktion gibt den Sinus eines *numerischen_Ausdrucks*, der im Bogenmaß angegeben werden muß, zurück.

• Details

Diese Funktion wird einfach genau berechnet, es sei denn, der *numerische_Ausdruck* wird mit doppelter Genauigkeit angegeben. In diesem Fall erfolgt auch die Berechnung mit doppelter Genauigkeit.

• Siehe auch

`COS, TAN, ATN`

• Beispiel

```
DEF Kosekans(a)=1/SIN(a)
```

SLEEP Befehl

• Syntax

```
SLEEP
```

• Wirkungsweise

Läßt das Programm auf ein Ereignis (Event) warten.

• Details

Dieser Befehl wird beendet, sobald irgendein Ereignis auftritt, inklusive Tastatureingaben und Ereignissen, die abgeschaltet oder unterbrochen wurden, z.B. MENU STOP. Aus diesem Grund wird SLEEP oft in ◆ Schleifen benutzt.
SLEEP sollte anstatt den wenig flexiblen GOTOs oder LOOPs benutzt werden, so daß auch andere Tasks in dem Betriebssystem laufen können, ohne daß der Warte-Task die ganze Prozessorzeit für sich beansprucht.

• Beispiel

```
DO
   IF MOUSE(0)<>0 THEN EXIT LOOP
   SLEEP
LOOP

' läuft in einer Schleife, bis die linke
' Maustaste gedrückt wurde.
```

SOUND Befehl

• Syntax

```
SOUND Frequenz,Dauer,[,[Lautstärke][,Stimme]]
SOUND WAIT
SOUND RESUME
```

• Wirkungsweise

Gibt einen Klang aus dem Lautsprecher wieder.

• Details

SOUND WAIT bewirkt, daß die nachfolgenden SOUND-Befehle zwischengespeichert werden, bis ein SOUND RESUME-Befehl ausgeführt wird; das erlaubt, zwei oder mehr Klänge für verschiedene Stimmen zu synchronisieren.
Frequenz gibt die Frequenz des Klanges in Hertz zwischen 20 und 15000 an. Dabei handelt es sich um eine Zahl einfacher Genauigkeit.

C	130,81	mittleres C	523,25
D	146,83	D	587,23
E	164,81	E	659,26
F	174,61	F	701,00
G	196,00	G	738,99
A	220,00	A	880,00
H	246,94	H	993,00
C	261,63	C	1046,50
D	293,66	D	1174,70
E	329,23	E	1318,50
F	349,23	F	1396,90
G	392,00	G	1568,00
Kammerton A	440,00	A	1760,00
H	493,88	H	1975,50

Dauer ist ein numerischer Ausdruck einfacher Genauigkeit zwischen 0 und 77. Die Zeit in Sekunden ist *Dauer*/18.2.
Lautstärke ist ein optionaler Integer-Parameter von 0 (kleinste Lautstärke) bis 255 (größte Lautstärke). Wird er weggelassen, wird ein Lautstärkewert von 127 benutzt.
Stimme ist ein optionaler Parameter, der dafür sorgt, daß der Amiga-Audiokanal benutzt wird. Die Grundeinstellung ist 0. Die Stimmen 0 und 3 benutzen den linken Kanal und 1 und 2 den rechten.

• Siehe auch

WAVE

• Beispiel

```
SOUND 440,9.1,255,2   ' lauter Kammerton A für
                      ' eine halbe Sekunde auf
                      ' dem rechten Kanal
```

SPACE$ Funktion

• Syntax

```
SPACE$(n)
```

• Wirkungsweise

Diese Funktion gibt eine Reihe von Leerstellen (ASCII 32) zurück, deren Anzahl durch *n* bestimmt wird.

• Details

n muß positiv sein und darf den Bereich einer Long-Integer-Zahl nicht überschreiten. Beim Druck von Leerstellen ist es besser, die SPC-Funktion zu verwenden.

• Siehe auch

```
STRING$
```

• Beispiel

```
DIM Name$(100)
FOR i=0 TO 100
   Name$(i)=SPACE$(30)
NEXT i
```

SPC Funktion

• Syntax

`SPC(n)`

• Wirkungsweise

Diese Funktion bewirkt, daß *n* Leerstellen in einem PRINT-Befehl übersprungen werden.

• Details

Im Gegensatz zu SPACE$ kann SPC nur in Zusammenhang mit PRINT und LPRINT benutzt werden. *n* muß positiv sein und im Bereich einer Long-Integer-Zahl liegen. Es ist nicht notwendig, ein Semikolon nach SPC anzugeben, da die Funktion nichts anderes als *n* Leerstellen erzeugt.
Normalerweise ist es das beste, SPACE$ dann zu benutzen, wenn man PRINT zu einem späteren Zeitpunkt durch einen anderen Befehl ersetzen will.

• Siehe auch

`PRINT, SPACE$, STRING$`

• Beispiel

```
LOCATE 10,1
INPUT "Name:",a$
LOCATE 10,1
PRINT SPC(5+LEN(a$))
LOCATE 10,1
INPUT "Adresse:",b$
```

SQR Funktion

• Syntax

SQR(*numerischer_Ausdruck*)

• Wirkungsweise

Diese Funktion gibt die Quadratwurzel eines *numerischen_Ausdruck* zurück.

• Details

Der *numerische_Ausdruck* muß größer oder gleich 0 sein. Normalerweise wird SQR mit einfacher Genauigkeit berechnet; hat der *numerische_Ausdruck* allerdings doppelte Genauigkeit, wird auch SQR doppelt genau berechnet.

• Beispiel

```
PRINT SQR(42)
PRINT SQR(42#)
```

Ergebnis:

```
6.480741
6.48074069840786
```

STATIC Befehl

• Syntax

`STATIC` *`Variablenliste`*

• Wirkungsweise

Dieser Befehl deklariert eine oder mehrere Variablen für Unterprogramme und Funktionsdefinitionen als lokal und sorgt dafür, daß deren Inhalt, auch wenn die Funktion oder das Unterprogramm verlassen und wieder aufgerufen wird, nicht gelöscht werden.

• Details

Dieser Befehl darf nur in Funktionsdefinitionen und Unterprogrammen benutzt werden.
Variablen in DEF FN-Funktionsdefinitionen sind normalerweise global. STATIC macht diese Variablen in der Funktion lokal.
Wir empfehlen, den STATIC-Befehl sogar dann zu benutzen, wenn dies als Grundeinstellung bereits definiert ist, da man dann nämlich die Variablenüberprüfungsoption (VARCHECKS) verwenden kann.

• Siehe auch

`DIM, LOCAL, SHARED, COMMON SHARED`

• Beispiel

```
test=10
PRINT FNbeispiel,test
DEF FNbeispiel
STATIC test ' ohne dies ist test global
test=5      ' hier wird test verändert
FNbeispiel=1/test
END SUB
```

STICK Funktion

• Syntax

```
STICK(n)
```

• Wirkungsweise

Diese Funktion gibt die x- und y-Position der beiden Joysticks zurück.

• Details

n kann folgende Werte haben:

0	gibt die x-Position von Joystick 1 zurück
1	gibt die y-Position von Joystick 1 zurück
2	gibt die x-Position von Joystick 2 zurück
3	gibt die y-Position von Joystick 2 zurück

Achtung! Durch das Lesen von Joystick 1 wird die Maus für die weitere Benutzung ausgeschaltet.

Für die x-Position gibt die Funktion folgende Werte zurück:
-1: rechts 0: Mitte 1: links

Für die y-Position gibt die Funktion folgende Werte zurück:
-1: unten 0: Mitte 1: oben

• Siehe auch

```
MOUSE,STRIG
```

• Beispiel

```
PSET (200,100)
DO
   XA% = STICK(2)
   YA% = STICK(3)
   PSET STEP (XA%,-YA%)  ' setze Pixel relativ
LOOP UNTIL STRIG(3)      ' bis Feuerknopf gedrückt
                         ' wird
```

STOP Befehl

• Syntax

```
STOP [Prozessrückgabecode]
```

• Wirkungsweise

Dieser Befehl beendet das Programm; alle Dateien werden geschlossen und die Kontrolle wird an das Betriebssystem zurückgegeben.

• Details

Wird kein Parameter angegeben, erscheint die Meldung „Taste drücken" am unteren Bildschirmrand, und es wird auf einen Tastendruck gewartet. Um dies zu verhindern, muß man eine -1 als Parameter übergeben. Wird ein anderer Parameter als -1 angegeben, handelt es sich dabei um einen AmigaDOS-Prozeßrückgabecode; dieser ist z.B. für Programme, die vom CLI aus gestartet werden, gedacht. Voreingestellt ist der Rückgabecode 0 (kein Fehler) oder der Fehlercode+300, falls ein Laufzeitfehler auftritt.

• Siehe auch

END

• Beispiel

```
IF FEXISTS(COMMAND$) THEN
  process COMMAND$
  STOP -1   ' kommt sofort zurück
ELSE
  PRINT "Kann ";COMMAND$;" nicht finden!"
  STOP      ' wartet auf einen Tastendruck
END IF
```

STR$ Funktion

• Syntax

STR$(*numerischer_Ausdruck*)

• Wirkungsweise

Diese Funktion gibt den Wert eines *numerischen_Ausdrucks* als String zurück.

• Details

STR$ ist das Gegenstück zu VAL.
Ist der *numerische_Ausdruck* ≥ 0, wird der String mit einer führenden Leerstelle zurückgegeben, anderenfalls beginnt er mit mit einem Minuszeichen. Der *numerische_Ausdruck* kann beliebigen numerischen Typs sein.

• Siehe auch

VAL, FORMATx$

• Beispiel

```
A%=4
B%=6
WRITE STR$(A%*B%)
```

Ergebnis:

```
" 24"
```

STRIG Funktion

• Syntax

```
STRIG(n)
```

• Wirkungsweise

Diese Funktion gibt den Status des Feuerknopfs des angegebenen Joysticks zurück.

• Details

n darf folgende Werte haben:

0 gibt -1 zurück, wenn der Feuerknopf von Joystick 0 seit dem letzten STRIG(0) gedrückt wurde, anderenfalls wird eine 0 zurückgegeben.

1 gibt -1 zurück, wenn der Feuerknopf von Joystick 0 gedrückt wurde, anderenfalls wird eine 0 zurückgegeben.

2 gibt -1 zurück, wenn der Feuerknopf von Joystick 1 seit dem letzten STRIG(2) gedrückt wurde, anderenfalls wird eine 0 zurückgegeben.

3 gibt -1 zurück, wenn der Feuerknopf von Joystick 1 gedrückt wurde, anderenfalls wird eine 0 zurückgegeben.

Achtung! Durch das Lesen von Joystick 1 wird die Maus für die weitere Benutzung ausgeschaltet.

• Siehe auch

```
STICK
```

• Beispiel

```
IF STRIG(3) = -1 THEN
  PRINT "Der Knopf an Joystick 2 wurde gedrückt"
END IF
' Joystick 2 heißt im Programm aus
' programmiertechnischen Gründen Joystick 1
```

STRING$ Funktion

• Syntax

STRING$(*m*,*n*)
STRING$(*m*,*String_Ausdruck*)

• Wirkungsweise

Diese Funktion gibt einen String, bestehend aus Zeichen mit dem ASCII-Code *n*, mit der Länge *m* zurück.

• Details

Wenn ein *String_Ausdruck* anstelle von *n* angegeben wurde, wird das erste Zeichen des *String_Ausdrucks* benutzt.

• Siehe auch

SPACE$

• Beispiel

```
PRINT STRING$(40,169)
```

Ergebnis:
©©

SUB...END SUB Befehl

• Syntax

```
SUB globaler_Name[(Parameterliste)][STATIC]
.
.
[EXIT SUB]
.
.
END SUB
```

• Wirkungsweise

Mit diesem Befehl wird ein Unterprogramm definiert.

• Details

globaler_Name muß ein Name sein, der nur einmal im ganzen Programm vorkommt. Es ist der Name des Unterprogramms.

Parameterliste ist die Parameterliste, die an das Unterprogramm übergeben wird. Das können Variablen-, Array-Namen, die mit Klammern () beendet werden, oder Zahlenwerte sein. Wird davor ein BYVAL angegeben, werden die Parameter immer als Wert übergeben; dadurch wird die Ausführung des Unterprogramms beschleunigt.

STATIC ist ein optionales Attribut, das anzeigt, daß das Unterprogramm nicht rekursiv ist, d.h. sich selbst nicht aufruft. In der Grundeinstellung haben Unterprogramme die Möglichkeit, rekursiv zu sein. Beim Schreiben ignoriert der Compiler dies.

EXIT SUB bewirkt, daß die Programmausführung wieder zu dem Teil des Programms zurückkehrt, von dem aus das Unterprogramm aufgerufen wurde. Es dient dazu, das Unterprogramm vor END SUB zu beenden.
Variablen können in einem Unterprogramm als LOCAL, SHARED oder STATIC deklariert werden. Näheres dazu findet man bei den entsprechenden Befehlsbeschreibungen.
Anwenderdefinierte Funktionen dürfen nicht in Unterprogrammen benutzt werden.

• Siehe auch

FUNCTION, STATIC, SHARED, LOCAL, EXIT, END, CALL

• Beispiel

```
CALL Fred(1,2)
.
.
SUB Fred(Eins%,Zwei%)
STATIC RekursivTiefe
   INCR RekursivTiefe
   IF RekursivTiefe > 4 THEN EXIT SUB
   PRINT Eins% Zwei%
   CALL Fred(Eins%*2,Zwei%*2)
END SUB
```

Ergebnis:

```
 1   2
 2   4
 4   8
 8   16
```

SWAP Befehl

• Syntax

```
SWAP Variable1, Variable2
```

• Wirkungsweise

Mit diesem Befehl lassen sich die Werte zweier Variablen miteinander vertauschen.

• Details

Beide Variablen müssen von genau demselben Typ sein und dürfen numerisch oder String sein.

• Siehe auch

```
LSET, RSET
```

• Beispiel

```
A% = 1
B% = 2
SWAP A%,B%
PRINT A%;B%
```

Ergebnis:

```
 2  1
```

SYSTAB Funktion

• Syntax

SYSTAB

• Wirkungsweise

Gibt die Adresse einer internen Systemtabelle zurück.

• Details

Diese Funktion ist für erfahrene Programmierer gedacht, die direkt auf Variablen des laufenden Systems zugreifen möchten. Auf die Tabelle sollte mit PEEK- und POKE-Befehlen zugegriffen werden.

Offset	*Größe*	*Beschreibung*
2	Wort	Pixelhöhe des Workbench-Bildschirms
4	Langwort	Zeiger auf den Intuition-Message-Port
8	Langwort	Zeiger auf die Workbench-Startmeldung (0 bei CLI)
12	Langwort	Zeiger auf Bildschirm 1 (0 wenn geschlossen)
16	Langwort	Zeiger auf Bildschirm 2 (0 wenn geschlossen)
20	Langwort	Zeiger auf Bildschirm 3 (0 wenn geschlossen)
24	Langwort	Zeiger auf Bildschirm 4 (0 wenn geschlossen)
28	Langwort	Zeiger auf den Zeichensatz *Topaz 8*
32	Byte	Anzahl der Ebenen des Workbench-Bildschirms
33	Byte	Icon-Typ; benutzt man beim Erstellen von Icons für Datendateien, voreingestellt ist 2 (=Projekt). Mit 0 wird die Icon-Datei deaktiviert und Dateioperationen, die .info-Dateien betreffen (z.B. KILL), werden verhindert.
34	Langwort	Zeiger, der beim Erzeugen von Icon-Dateien benutzt wird; normalerweise 0 (für das Default-Icon). Wird das verändert, wird dieser Offset als Zeiger auf eine Bildstruktur verwendet.
38	Byte	ASCII-Zeichen des Währungssymbols, das bei PRINT USING für auf $-basierendes Formatieren benutzt wird. Die Voreinstellung ist ASC("$").
40	Langwort	Größe des INPUT#-Zeilenpuffers
44	Langwort	Bei Benutzung der HEAPDYNAMIC-Option enthält dies die maximale Größe des Arbeitsspeichers, die alloziert werden kann, bevor die nächste Garbage-Collection stattfindet.

Mit Ausnahme der Offsets 2 bis 32 sind diese Einträge nur zum Lesen gedacht und sollten nicht mit POKE verändert werden. Es wird nicht garantiert, daß undokumentierte SYSTAB-Offsets von einer Compiler-Version zur nächsten gleich bleiben; aber die oben beschrieben sind garantiert.

• Siehe auch

PEEKL, PEEKW, PEEKB, LINE INPUT#, PRINT USING

• Beispiel

```
w%=PEEKW(SYSTAB): h%=PEEKW(SYSTAB+2)
SCREEN 1,w%,h%,PEEKB(SYSTAB+32),2
IF PEEKL(SYSTAB+8) THEN
  a$="Von der Workbench"
ELSE
  a$="Vom CLI"
END IF
WINDOW 1,a$,(20,20)-(300,100),,1
```

TAB Funktion

• Syntax

`TAB(n)`

• Wirkungsweise

Diese Funktion bewirkt, daß die Ausgabeposition in die *n*-te Spalte springt.

• Details

Befindet sich die aktuelle Ausgabepostion bereits weiter als *n*, springt die Ausgabeposition in die *n*-te Spalte der nächsten Zeile. Ist *n* größer als die mögliche Ausgabebreite, springt die Ausgabe auf die Position *n* MOD WIDTH. Ist *n* kleiner als 1, wird die Ausgabeposition ebenfalls 1. TAB darf nur im Zusammenhang mit PRINT- und LPRINT-Befehlen benutzt werden.

• Siehe auch

`PRINT, LPRINT, SPC`

• Beispiel

```
' Bildschirm mit 80 Zeichen Breite
PRINT TAB(-10) "minus Zehn"
PRINT TAB(175) "Einhundertfünfundsiebzig"
PRINT TAB(10) "Zehn"
```

Ergebnis:

```
minus Zehn
                    Einhundertfünfundsiebzig
          Zehn
```

TAGLIST Befehl

• Syntax

`TAGLIST Adress_Ausdruck, Ausdrucksliste`

• Wirkungsweise

Dieser Befehl baut eine Taglist auf, um damit das Amiga-Betriebssystem aufzurufen.

• Details

Der erste Parameter, *Adress_Ausdruck*, ist die Speicheradresse, an der die Taglist gespeichert wird. Die Elemente der Taglist müssen entweder numerische (die zu Long-Integer-Zahlen konvertiert werden) oder String-Ausdrücke sein. Letztere werden zu Null-terminierten Strings (wie in C) konvertiert und ein Zeiger darauf benutzt.
Taglists werden von den meisten neuen Features des Betriebssystems benutzt, die mit Einführung der Workbench 2 bzw. höherer Versionen verhindern sollen, daß man auf komplizierte Strukturen oder Betriebssystemroutinen mit vielen Parametern zurückgreifen muß. Sie bestehen aus 0 oder mehreren Paaren, die jeweils ein Tag enthalten, das angibt, welchen Parameter man setzen will, und einem Wert, den man dem Tag übergibt. Die Liste wird immer mit TAG_END& abgeschlossen, das den Wert 0& hat. Konstanten für die verschiedenen Taglists werden in den .bh-Dateien unterstützt, die MaxonBASIC zur Verfügung stellt.
Achtung! String-Werte werden ungültig, sobald eine Garbage Collection stattfindet; deshalb sollte man sicherstellen, daß genug Speicherplatz auf dem Heap für alle benötigten Strings vorhanden ist, bevor man die Taglist erstellt. Nehmen wir an, man will eine Taglist aus Strings erstellen, die x Bytes lang sind, dann kann man dies folgendermaßen überprüfen:

```
IF FRE(0)< x*2 THEN junk&=FRE("")
```

Das Zeile-geht-weiter-Zeichen (_) kann optimal benutzt werden, so daß nicht alle Tags in eine Zeile Quellcode gequetscht werden müssen. Man muß sicherstellen, daß das Array, in dem die Taglist gespeichert werden soll, groß genug ist, um alle Elemente aufzunehmen; der Compiler kann einem diese Kontrollarbeit nicht abnehmen.

• Beispiel

```
' Dieses Beispiel benötigt Workbench 2
REM $include intuition.bh
LIBRARY OPEN "intuition",37
CONST TAG_END&=0,TRUE&=1
DIM temptag&(20)
TAGLIST VARPTR(temptag&(0)), _
   WA_Title&,  "Klicken Sie zum Schließen auf das
                                andere Fenster", _
   WA_Width&,  400&, _
   WA_Height&, 100&, _
   WA_CloseGadget&,   TRUE&, _
   WA_IDCMP&,  IDCMP_CLOSEWINDOW&, _
   TAG_END&

mein_Fenster& = OpenWindowTagList&(0,VARPTR
                                    (temptag&(0)))
SLEEP
CloseWindow mein_Fenster&
```

TAN Funktion

• Syntax

`TAN(numerischer_Ausdruck)`

• Wirkungsweise

Diese Funktion liefert den Tangens eines *numerischen_Ausdrucks*, der im Bogenmaß angegeben werden muß.

• Details

Der Tangens wird normalerweise mit einfacher Genauigkeit berechnet. Ist allerdings der *numerische_Ausdruck* doppelt genau, wird auch der Rückgabewert mit doppelter Genauigkeit berechnet.

• Siehe auch

`ATN, SIN, COS`

• Beispiel

```
PRINT TAN(0.5)
```

TIME$ Funktion

• Syntax

```
TIME$
```

• Wirkungsweise

Die Funktion gibt die aktuelle Systemzeit zurück.

• Details

Es wird ein String mit folgendem Format zurückgegeben:

hh:mm:ss	*hh* sind die Stunden im 24-Std.-Format (00-23) *mm* sind die Minuten (00-59) *ss* sind die Sekunden (00-59)

• Siehe auch

```
DATE$, TIMER
```

• Beispiel

```
PRINT TIME$
```

Ergebnis:

```
20:43:56
```

TIMER Befehl

• Syntax

```
TIMER {ON|OFF|STOP}
```

• Wirkungsweise

Modifiziert die Timer-Ereignis-Überwachung (Timer Event Trapping), um die Wirkungsweise des ON...TIMER-Befehls zu ändern.

• Details

TIMER ON wird zum Einschalten der Timer-Überprüfung benutzt. Dadurch wird bewirkt, daß ON TIMER-Befehle aktiviert werden, sobald ein Timer-Ereignis auftritt.
Nach einem TIMER OFF-Befehl bleibt die Timer-Überprüfung solange ausgeschaltet, bis ein anderer TIMER ON-Befehl ausgeführt wird.
TIMER STOP bewirkt, daß Timer-Ereignisse solange zwischengespeichert werden, bis ein TIMER ON-Befehl auftritt. Das kann ganz nützlich sein, wenn man zeitkritischen Code ausführen will.

• Siehe auch

`ON...TIMER, TIMER-Funktion`

• Beispiel

```
ON TIMER(60) GOSUB Einmal_pro_Minute
TIMER ON
```

TIMER Funktion

• Syntax

```
TIMER
```

• Wirkungsweise

Diese Funktion gibt die Anzahl der Sekunden seit Mitternacht (0 Uhr) als Zahl mit einfacher Genauigkeit zurück.

• Details

Diese Funktion kann als Ausgangszahl für RANDOMIZE benutzt werden. Sie gibt Werte mit einer Genauigkeit von 200stel Sekunden zurück. Sie kann z.B. für Zeitmeßprogramme genutzt werden. Nachtschichtprogrammierer sollten darauf achten, daß das unten angegebene Beispiel nicht korrekt funktioniert, wenn zwischen den Aufrufen Mitternacht liegt!

• Siehe auch

```
TIME$
```

• Beispiel

```
t!=TIMER
call Hauptprogramm
t!=TIMER-t!
PRINT "Es dauerte";t!;"Sekunden"
```

TRANSLATE$ Funktion

• Syntax

`TRANSLATE$(String_Ausdruck)`

• Wirkungsweise

Konvertiert einen String aus dem Englischen in Phonem-Folgen, die unter älteren Versionen des Betriebssystems für den SAY-Befehl benutzt werden können.

• Details

Dieser Befehl benutzt die Translator-Library, die ab Workbench 2.1 aus dem Betriebssystem genommen wurde. Entwickler vertreiben diese nicht mehr, so daß wir deren Benutzung nicht empfehlen können; Besitzer neuerer Rechner (z.B. A1200 und A4000) können diese Library gar nicht mehr haben.
Achtung! Unsere Standard-Disketten-Installation kopiert *nicht* die translator.library, das narrator.device und den Speak-Handler auf die Boot-Diskette, die von ihr erzeugt wird; man muß diese Dateien selbst auf die Boot-Diskette kopieren und den MonAm-Debugger von der Diskette löschen.

• Siehe auch

`SAY`

• Beispiel

```
SAY TRANSLATE$("Hello John")
```

TRON,TROFF Befehle

• Syntax

```
TRON
TROFF
```

• Wirkungsweise

Diese Befehle erlauben, ein Programm während der Laufzeit zeilenweise auszuführen.

• Details

Nach einem TRON-Befehl werden alle Programmzeilen auf dem Bildschirm in der Reihenfolge ausgegeben, wie sie ausgeführt werden; das kann mit TROFF abgeschaltet werden.
Zeilennummern werden in eckigen Klammern ausgegeben.
Die Benutzung von TRON in einem Programm, verlangsamt die Programmausführung massiv, auch noch nach einen TROFF-Befehl. Aus diesem Grund sollte man TRON aus dem Quellcode nehmen, sobald das Programm fertiggestellt ist.

• Beispiel

```
1000 TRON
1010 PRINT "Problematischer Teil des Programms"
1020 INPUT a
1030 PRINT "Input:";a,"Ergebnis:";fnTest(a)
1040 TROFF
```

UBOUND Funktion

• Syntax

`UBOUND(Array_Name[,Dimension])`

• Wirkungsweise

Diese Funktion gibt den größtmöglichen Index eines bestimmten Arrays (und optional auch der Dimension) zurück.

• Details

Dimension gibt die Dimension des Arrays an, für das die Obergrenze angegeben werden soll. Wird *Dimension* nicht angegeben, ist die Grundeinstellung 1.

• Siehe auch

`LBOUND, DIM`

• Beispiel

```
DIM Array%(10,20,30)
PRINT UBOUND(Array%,2)
```

Ergebnis:

```
 20
```

UCASE$ Funktion

• Syntax

`UCASE$(String_Ausdruck)`

• Wirkungsweise

Diese Funktion gibt einen String mit beliebigen alphabetischen Zeichen in Großbuchstaben aus.

• Details

Zeichen im Bereich zwischen „a" und „z" inklusive werden je nach Erfordernis in „A" bis „Z" gewandelt. Alle anderen Zeichen inklusive Sonderzeichen und Zeichen mit Akzenten werden nicht beeinflusst.

• Siehe auch

`LCASE$`

• Beispiel

```
' Das Array muß zuvor übergeben werden.
' 1 definiert die Anzahl der Dimensionen
FUNCTION finde(BYVAL a$,Array$(1))
STATIC i,f
f=0
FOR i=LBOUND(Array$) TO UBOUND(Array$)
   IF UCASE$(f$)=Array$(i) THEN f=i: EXIT FOR
NEXT i
finde=f
END FUNCTION
```

VAL Funktion

• Syntax

`VAL(String_Ausdruck)`

• Wirkungsweise

Diese Funktion gibt den numerischen Wert eines *String_Ausdrucks* zurück.

• Details

Diese Funktion überprüft einen *String_Ausdruck* nach einer Zeichenfolge, die als Zahl interpretiert werden kann. VAL hört auf zu lesen, sobald es ein Zeichen findet, das nicht als Zahl erkannt werden kann. Führende Leerstellen, Tabulatoren und Line-Feeds werden ignoriert. VAL gibt eine Zahl doppelter Genauigkeit zurück, kann aber natürlich auch einer Variablen mit geringerer Genauigkeit zugewiesen werden.

• Siehe auch

`STR$`

• Beispiel

```
Adresse$=" 65760 Eschborn"
Wert%=VAL(Adresse$)   ' Ergebnis ist 65760
```

VARPTR Funktion

• Syntax

VARPTR(*Variablenname*)
oder
VARPTR(*#Kanal*)

• Wirkungsweise

Die Funktion gibt die Speicheradresse einer Variablen mit dem Namen *Variablenname* oder des Ein-/Ausgabepuffers eines *Kanals* zurück.

• Details

Die Rückgabeadresse ist eine Long-Integer-Zahl. VARPTR kann mit jedem numerischen Variablentyp benutzt werden; zum Finden einer String-Adresse muß man SADD benutzen. Um die Adresse eines Arrays zu erhalten, muß man das erste Element des Arrays angeben.
Man sollte VARPTR erst dann benutzen, wenn man die Rückgabeadresse verwenden will; lokale numerische Variablen und alle Strings und Arrays haben nicht dieselbe Speicheradresse während des Programmablaufs.
VARPTR mit einer Kanalnummer als Parameter gibt die Adresse des Ein-/Ausgabepuffers eines bereits geöffneten Kanals zurück. Ist der Kanal noch nicht geöffnet, wird eine 0 zurückgegeben. Man kann auch überprüfen, ob ein MaxonBASIC-Fenster geöffnet ist; dazu nimmt man die Kanalnummer 257+i für das Fenster i. Wird ein Wert ungleich 0 zurückgegeben, ist das Fenster geöffnet; bei 0 ist es geschlossen.
VARPTR gibt immer eine gerade Adresse zurück.
VARPTR wird auch in Funktionsdefinitionen benutzt, um Variablenparameter (im Gegensatz zu Wertparametern) zu definieren.

• Siehe auch

SADD, VARPTRS

• Beispiel

```
Zahl% = 7
Adresse& = VARPTR(Zahl%)
PRINT HEX$(Adresse&)
```

Ergebnis:
6A558

VARPTRS Funktion

• Syntax

`VARPTRS(Unterprogrammname)`

• Wirkungsweise

Diese Funktion gibt die Speicheradresse eines Unterprogramms *Unterprogrammname* zurück.

• Details

Die Adresse wird als Long-Integer-Zahl zurückgegeben. Diese Funktion kann in Verbindung mit CALLS benutzt werden. Das Unterprogramm darf keine Parameter haben.
Achtung! Diese Funktion ist nur für fortgeschrittene Programmierer gedacht.

• Beispiel

```
IF von_Disk THEN
   lese_Routine&=VARPTRS(Disk_lesen)
ELSE
   lese_Routine&=VARPTRS(RAM_lesen)
END IF
```

WAVE Befehl

• Syntax

```
WAVE Stimme,{Wave_Array|SIN}
```

• Wirkungsweise

Bestimmt die Wellenformen, die vom SOUND-Befehl benutzt werden.

• Details

Stimme gibt den Audiokanal an, der benutzt werden soll.
Ist der zweite Parameter SIN, gibt die Sinus-Funktion die Wellenform an. Wird ein Array benutzt, sollte es mindestens 256 Elemente haben. Die Elemente in dem Array müssen im Bereich zwischen -128 und 127 liegen.

• Siehe auch

SOUND

• Beispiel

```
WAVE 1,SIN
WAVE 2,Wave_Tabelle
```

WHILE...WEND Befehl

• Syntax

```
WHILE Bedingung
.
.
[Befehle]
.
.
WEND
```

• Wirkungsweise

Eine Reihe von Befehlen wird solange in einer Schleife ausgeführt, bis *Bedingung* FALSE wird.

• Details

Solange *Bedingung* TRUE ist, werden die Befehle zwischen WHILE und WEND ausgeführt.
WHILE...WENDs dürfen ineinander verschachtelt werden; ein WEND überprüft das letzte WHILE.
Man muß darauf achten, daß man eine WHILE...WEND-Schleife nicht von innen heraus beginnt (z.B. durch einen GOTO-Sprung in die Schleife).
DO...LOOP- und REPEAT-Schleifen sind üblicher als WHILE-Schleifen.

• Siehe auch

DO..LOOP, REPEAT

• Beispiel

```
A%=0
WHILE A%<10
   INCR A%
   PRINT A%;
WEND
```

Ergebnis:

```
 1  2  3  4  5  6  7  8  9  10
```

WIDTH Befehl

• Syntax

```
WIDTH [#Kanalnummer,] Breite [,Tabulator]
WIDTH LPRINT Breite[,Tabulator]
```

• Wirkungsweise

Diese Befehle bestimmen die Zeilenbreite einer angegebenen Datei, des Bildschirms oder eines Druckers.

• Details

Folgende Parameter sind mit diesem Befehl verwendbar:

Breite
Hiermit wird die Breite für die Bildschirmausgabe bestimmt.

#Kanalnummer,Breite
Hiermit wird die Breite für die Ausgabe in einen Datenkanal mit einer angegebenen *Kanalnummer* mit einer angegebenen Breite definiert. Das #-Zeichen muß unbedingt verwendet werden.

Tabulator
Bestimmt die Länge eines Tabulators; voreingestellt sind 14 Stellen.

Benutzt man WIDTH in Zusammenhang mit LPRINT, kontrolliert es die *Breite* und den *Tabulator* jeglicher Ausgabe auf einen Drucker. Wird eine *Breite* von 255 angegeben, ist eine unbegrenzte Breite eingestellt, d.h. MaxonBASIC fügt dann kein Line-Feed während der Ausgabe ein.

• Beispiel

```
WIDTH 76        ' ein 80-Zeichen-Bildschirm
WIDTH 61        ' ein 60-Zeichen-Bildschirm
WIDTH LPRINT 40,8 ' setzt die Druckerausgabe auf
                  ' 40 Spalten und eine
                  ' Tabulatorlänge von 8
```

WINDOW Befehle

```
WINDOW ID[,[Titel_String][,(x1,y1)-(x2,y2)]
                          [,[Typ][,Bildschirm_ID]]]
WINDOW CLOSE ID
WINDOW OUTPUT ID
```

• Wirkungsweise

Erzeugt und schließt Fenster; WINDOW OUTPUT wechselt das aktuelle Fenster.

• Details

Die meisten Befehle erfordern eine Fenster-*ID*. Dabei handelt es sich um eine Zahl von 1 bis 12 inklusive, die dazu benutzt wird, jedes Fenster zu identifizieren. Wird ein Programm gestartet, öffnet es ein Fenster mit der *ID* 1 (voreingestellt) und den Ausmaßen 640 x 200 Pixel.
Alle PRINT-, INPUT- und Grafikbefehle benutzen dieses Fenster, *aktuelles Fenster* genannt, bis sie ein anderes zugewiesen bekommen.

WINDOW CLOSE *ID*
Hiermit wird ein Fenster mit der angegebenen *ID* geschlossen.

WINDOW OUTPUT *ID*
Hiermit wird das angegebene Fenster zum aktuellen gemacht. Alle PRINT-, INPUT- und Grafikbefehle benutzen nun diese Fenster.

WINDOW
Hierbei handelt es sich um die komplexeste Form des WINDOW-Befehls; er erzeugt ein Fenster anhand der angegebenen Spezifikationen. Alle Parameter (mit Ausnahme von *ID*) sind optional, und wenn sie weggelassen werden, wird das bereits existierende Fenster zum aktiven gemacht.
Titel_String wird für den Titelbalken des Fensters verwendet.
Die Koordinaten (*x1,y1*) geben die Position der linken oberen und (*x2,y2*) die der rechten unteren Ecke des Fenster*inhalts* an. Die voreingestellte Größe des Fensterrechtecks ist entweder ein Fenster, das auf einem 640 x 200-Workbench-Bildschirm oder der vollen Bildschirmgröße basiert.
Der *Typ*-Parameter bestimmt die Fensterelemente, die durch Addieren der verschiedenen Element*typen* je nach Erfordernis bestimmt werden. Folgende Elemente sind möglich:

Wert *Bedeutung*

1 Die Fenstergröße kann mit der Maus und dem Größenelement (Sizing-Gadget) verändert werden.
2 Das Fenster kann mit dem Titelbalken herumbewegt werden.
4 Das Fenster kann in Relation zu anderen Fenstern mit den normalen Fensterelementen herumbewegt werden.
8 Das Fenster kann durch Klicken auf das Schließelement (Close-Gadget) geschlossen werden [wird nur erkannt, wenn die Ereignisüberprüfung (Event Checking) an ist].
16 Der Inhalt des Fensters erscheint wieder, nachdem das Fenster durch andere Fenster überdeckt wurde. In Verbindung mit *Typ 1* reserviert BASIC genug Speicherplatz für das Fenster, daß es für eine volle Bildschirmgröße genügt.
32 Fenster ohne ‚gimmezerozero'. Das macht das Fenster-Handling zwar deutlich schneller, hat aber den Nachteil, daß nichts das Zeichnen über den Fensterrand hinaus verhindert.
64 Backdrop-Window; ein Fenster, das immer im Hintergrund auf dem Bildschirm angezeigt wird, wie z.B. das Disketten-Icon auf der Workbench. Diese Fensterart macht in der Verbindung mit *Typ 32* und *Typ 128* und auf anderen Bildschirmen als der Workbench am meisten Sinn.
128 Randloses Fenster; es wird kein Rand um dieses Fenster gezeichnet; am besten benutzt man es in Verbindung mit *Typ 32* und *Typ 256*.
256 Wird auch Fenster mit „sensiblen" Koordinaten genannt; benutzt man diese Art von Fenster wird die Interpretation von (x2,y2) so verändert, daß x2 die komplette Fensterbreite und y2 die Höhe des Fensters sind.

Die *Typen 32-256* sind Maxon BASIC-Erweiterungen. Es ist einiges an Erfahrung notwendig, um genau den Fenstertyp zu erhalten, den man erhalten möchte. Die Voreinstellung ist 31.
Die *Bildschirm-ID* bezieht sich auf einen Bildschirm, der mit dem SCREEN-Befehl erzeugt wurde und eine *ID* von 1 bis 4 oder -1 für den Workbench-Bildschirm (Voreinstellung) haben sollte.

• Siehe auch

SCREEN, WINDOW-Funktion

• Beispiel

```
SCREEN 1,640,200,2,2
WINDOW 1,,,16+32+64+128+256,1
LOCATE 10,50
PRINT "Ich bin nicht die Workbench!"
WINDOW 2,"Gewöhnliches Fenster",(0,20)-
                                    (320,100),,1
```

WINDOW Funktion

• Syntax

```
WINDOW(n)
```

• Wirkungsweise

Mit dieser Funktion kann man wichtige Informationen über Fenster erhalten, die für einige Anwendungen von Nutzen sein können. Der Parameter *n* liefert folgende Ergebnisse:

Wert	*Ergebnis*
0	Die Fenster-ID des aktiven Fensters oder 0, wenn das aktive Fenster nicht zum aktuellen Programm gehört.
1	Die Fenster-ID des aktuellen Ausgabefensters
2	Die Breite des aktuellen Ausgabefensters
3	Die Höhe des aktuellen Ausgabefensters
4	Die x-Koordinate, an der das nächste Zeichen ausgegeben wird
5	Die y-Koordinate, an der das nächste Zeichen ausgegeben wird
6	Die maximal legale Farbe des aktuellen Ausgabefensters
7	Ein Long-Integer-Zeiger auf das Intuition-Fenster des aktuellen Ausgabefensters. Wurde das Fenster geschlossen, wird eine 0 zurückgegeben.
8	Ein Zeiger auf den RastPort des aktuellen Ausgabefensters (oder 0, wenn er geschlossen wurde)
9	Gibt die Fenster-ID des letzten Fensters zurück, das vom Anwender zu schließen versucht wurde. Dient zur Nutzung in ON...CLOSE-Unterroutinen.

• Details

Nähere Informationen zu Fensterzeigern und RastPorts kann man in der Technischen Dokumentation unter Intuition und der Graphics-Library finden.
Ist das Fenster geschlossen, liefert der Compiler für *n* von 0 bis 6 willkürliche Ergebnisse.

• Siehe auch

CLOSE, ON...CLOSE, WINDOW-Befehl

WRITE Befehl

• Syntax

```
WRITE [Ausdrucksliste]
```

• Wirkungsweise

Dieser Befehl gibt die Daten, die in der *Ausdrucksliste* angegeben sind, auf dem Bildschirm aus.

• Details

Dieser Befehl verhält sich ähnlich wie PRINT. Wird die *Ausdrucksliste* weggelassen, wird ein Line-Feed auf dem Bildschirm ausgegeben. Alle Ausdrücke müssen durch Kommata getrennt werden. Die Kommata werden ebenfalls ausgegeben. Man kann auch Semikolons anstatt der Kommata verwenden; die werden allerdings auch als Kommata ausgegeben.
Gibt man Strings mit WRITE aus, werden diese in Anführungszeichen gesetzt. Bei Zahlen werden – im Gegensatz zu PRINT – keine führenden oder nachfolgenden Leerstellen ausgegeben.
Am Ende eines WRITE-Befehls wird immer ein Line-Feed ausgegeben.

• Siehe auch

```
PRINT, WRITE#, INPUT
```

• Beispiel

```
A% = 24
B$ = "STRING!"
PRINT A%,B$
WRITE A%,B$
```

Ergebnis:

```
 24          STRING!
24,"STRING!"
```

WRITE# Befehl

• Syntax

```
WRITE #Kanalnummer, Ausdrucksliste
```

• Wirkungsweise

Dieser Befehl schreibt Daten, die durch eine *Ausdrucksliste* bestimmt werden, in eine sequentielle Datei.

• Details

WRITE# unterscheidet sich von PRINT# in derselben Art und Weise wie WRITE von PRINT. WRITE# hat den Vorteil, daß die Daten mit seinem Äquivalentbefehl INPUT# zurückgelesen werden können; dies ist bei PRINT nicht der Fall.
Weitere Informationen findet man unter PRINT# und WRITE.

• Siehe auch

```
PRINT#, WRITE, INPUT#
```

• Beispiel

```
OPEN f$ FOR OUTPUT AS #1
FOR i=1 TO 10
   WRITE #1,i,CHR$(i+"a"%)
NEXT i
CLOSE #1
OPEN f$ FOR INPUT AS #1
WHILE NOT EOF(1)
   INPUT #1,a,a$
   PRINT a,a$
WEND
CLOSE #1
```

Tutorial

Im Folgenden werden Sie Ihr erstes Programm mit MB erstellen. Sie werden sehen, dies geht sehr schnell und ganz einfach. Sie werden dieses Programm starten und das erste Ergebnis auf dem Bildschirm bewundern.
Danach werden weitere Grundlagen und Grundbegriffe von BASIC und MB erläutert. Alle BASIC-Befehle von MB finden Sie alphabetisch sortiert im Kapitel "Befehlsübersicht".
Dieses Handbuch kann keine komplette Einführung in die Sprache BASIC sein. Vermittelt werden in erster Linie die Grundzüge. Zusammen mit der Befehlsübersicht können Sie sich den Zugang zu dieser umfangreichen und leistungsfähigen Sprache verschaffen. Wer schon einmal mit BASIC oder einer anderen Programmiersprache wie z.B. PASCAL oder C programmiert hat, oder sich im Freundeskreis Anregungen und Hilfe holen kann, wird damit auskommen. Einsteigern in die Programmierung wird jedoch empfohlen, sich ein Einstiegsbuch über die Sprache BASIC zu besorgen und dies durchzuarbeiten.

Doch beginnen wir endlich mit dem Programmieren in BASIC, damit Sie sehen können, wie Sie damit zurechtkommen. Sicherlich versetzen Sie die ersten Programmerfolge in eine Hochstimmung und Sie beginnen, mit den Befehlen und Möglichkeiten dieser schönen Programmiersprache zu experimentieren. In der Befehlsübersicht finden Sie zudem zu vielen Befehlen ein kurzes Beispiel, das den Einsatz kurz erläutert. Probieren Sie alles einfach einmal aus, Sie werden überrascht sein, wie einfach es ist ...

Das erste Programm

Ein erstes Programm soll die prinzipielle Bedienung von MB zeigen, damit Sie sofort einsteigen können.

1. Starten Sie das Programm MB durch Doppelklicken auf das Icon.

Nach einer kurzen Ladezeit erscheint MB mit einem leeren Editorfenster. Ein BASIC-Programm besteht aus mehreren Anweisungen oder Befehlen. Alle BASIC-Befehle entsprechen englischen Wörtern. In unserem Beispiel bedeutet PRINT "drucken" oder "ausgeben". BEEP heißt "Piep" und bewirkt normalerweise ein Piepsen des Computerlautsprechers. Bei MB wird jedoch der Bildschirm kurz aufblitzen.

2. Geben Sie nun die folgenden Zeilen nacheinander ein:

```
PRINT "Hallo Welt"
BEEP
```

3. Wählen Sie nun im Menü "PROGRAMM" den Punkt "RUN".

MB bereitet nun das Programm vor und startet es danach. Auf dem Bildschirm muß nun die Zeile "Hallo Welt" erscheinen, danach blinkt der Bildschirm kurz auf.

Wichtig: Sollte dies nicht der Fall sein, so haben Sie vermutlich bei der Eingabe der Zeilen unter 2. einen Fehler gemacht. Kontrollieren Sie alle Schritte ab 2. erneut.

Das Aussehen eines BASIC-Befehls

Eine BASIC-Anweisung kann aus einem Befehl bestehen (z.B. BEEP) oder, wie in den meisten Fällen, ein Befehl, gefolgt von einem Argument (z.B. PRINT). Das Argument liefert dem Befehl den notwendigen Inhalt (z.B. PRINT "Hallo Welt").

Das Aussehen von BASIC-Programmen

Wie bereits erwähnt, besteht ein BASIC-Programm aus einer Ansammlung von englischen Befehlen oder Anweisungen. Dabei ist es egal, ob die Befehle groß, klein oder gemischt geschrieben werden. Auch die Anzahl der Leerzeichen und Tabulatoren zwischen Befehlen oder Befehl und Argument ist frei wählbar.
Mehrere Befehle können sowohl untereinander, als auch in einer Zeile stehen. Sie werden dann durch einen Doppelpunkt voneinander getrennt. Da MB eine moderne BASIC-Variante ist, benötigt es keine Zeilennummern am Anfang der Zeile, wie dies bei älteren BASIC-Dialekten der Fall war.
Wir kommen auf die Struktur von BASIC-Programmen noch einmal zurück, wenn einige weitere Elemente dieser Sprache besprochen wurden, denn sonst wird der Sinn nicht klar werden.

Variablen - in ständiger Veränderung

Eine Variable ist praktisch ein Platzhalter, der zu einer bestimmten Zeit einen bestimmten Wert hat (z.B. 1), aber zu einer anderen Zeit auch einen anderen (z.B. 111). Der Wert kann also konstant sein (z.B. immer 2) oder sich verändern (z.B. 1, 2, 3 ...). Es stellt sich die Frage nach dem Sinn einer Variablen, könnte man den Wert doch gleich direkt angeben.

```
a=1
print a
```

Dies ist das Gleiche wie:

```
print 1
```

In beiden Fällen wird die Zahl "1" auf dem Bildschirm ausgegeben. Wieso also dieser Aufwand? Der große Vorteil einer Variable ist die Veränderlichkeit. Es ist dadurch möglich, etwas eine bestimmte Anzahl zu wiederholen, wie dies im nächsten Punkt beschrieben wird.

Anmerkung: Im klassischen BASIC wird die Zuweisung a=1 mit dem Befehl LET eingeleitet:

```
LET a=1
```

In MB ist der Zusatz LET nicht notwendig, er kann jedoch verwendet werden.

Die x-te Wiederholung

Das schöne an einer Programmiersprache ist, daß man bestimmte Aktionen beliebig oft wiederholen kann. Ein einfaches Beispiel soll dies verdeutlichen:

```
FOR a=1 TO 10 DO
   PRINT a
NEXT a
```

Dieses Programm wird die Zahlen von 1 bis 10 untereinander ausgeben. Testen Sie dieses Beispiel nun einmal, indem Sie die drei Zeilen in dem Editorfenster von MB eingeben und "RUN" im Menü "COMPILER" wählen.

Die erste Anweisung bestimmt, daß etwas wiederholt werden soll, und zwar so lange, bis die Variable "a" von "1" bis "10" hochgezählt wurde. Die zweite Zeile gibt den Inhalt der Variablen "a" aus, also die Zahlen 1 bis 10. Die dritte Zeile bildet das Ende der Anweisung von Zeile 1. Sie sagt, daß nun das nächste "a" an der Reihe ist.

Gehen wir das Programm noch einmal durch. Wir beginnen bei der ersten Zeile. Die Variable "a" bekommt den Wert "1" zugeordnet. Danach folgt die Anweisung "PRINT a". Damit wird der Wert von "a", also "1" ausgegeben. Die Anweisung "NEXT a" bestimmt, daß nun das nächste "a" an der Reihe ist. Es folgt nun ein Rücksprung in Zeile 1. Die Variable "a" wird um einen Wert weitergezählt und hat nun den Wert "2". Dieser Wert wird nun wieder mit "PRINT a" ausgegeben. Die Anweisung "NEXT a" bewirkt wieder einen Sprung in die 1. Zeile.

Wie lange wird das wohl so gehen? Genau so lange, wie dies in der ersten Programmzeile angegeben wurde. Hier steht, daß es bis 10 (englisch: TO 10) gehen soll. Nach dem 10. Durchlauf ist dieses Programm beendet.

Weitere Befehle zum Aufbau von "Schleifen" (so nennt man diese Art der Wiederholungsanweisung) sind

```
DO ... LOOP UNTIL
WHILE ... END WHILE
REPEAT ... END REPEAT
```

Sie finden die Erklärungen zu diesen Schleifenanweisungen in der Befehlsübersicht.

Anmerkung: Um ein übersichtliches Programm zu erhalten, werden die Anweisungen, die innerhalb der Schleife stehen um einige Zeichen nach rechts eingerückt. Dadurch sieht man sofort, welche Anweisungen innerhalb der Schleife bearbeitet, also mehrfach ausgeführt werden und welche Anweisungen danach, also wenn die Schleife beendet wurde, an der Reihe sind.

```
PRINT "Dies geschieht vor der Schleife"
FOR a=1 to 10
  PRINT "Jetzt sind wir in der Schleife, Durch-
gang Nummer: ";
  PRINT a
NEXT a
PRINT "Die Schleife ist jetzt beendet und es
geht weiter ..."
```

Testen Sie nun auch dieses Beispiel, indem Sie die Zeilen eingeben und das Programm mit RUN starten (siehe oben).

Entscheidungen

Neben der Schleife ist die Entscheidung und Fallunterscheidung ein wesentlicher Punkt eines Programms. Die Entscheidung sieht dabei folgendermaßen aus: Falls (englisch: IF) etwas so ist, dann (englisch: THEN) tue dies, sonst (englisch: ELSE) tue etwas anderes. In BASIC sieht es dann z.B. so aus:

```
IF a=1 THEN
   PRINT "a ist eins"
ELSE
   PRINT "a ist nicht eins"
END IF
```

Es kann natürlich auch vorkommen, daß man nicht wissen will was sonst (ELSE) passiert. Dann genügt die verkürzte Form:

```
IF a=1 THEN
   PRINT "a ist eins"
END IF
```

Wenn es mehr als zwei Möglichkeiten gibt, dann kann die IF-THEN-ELSE-Anweisung auch umfangreicher aufgebaut werden:

```
IF a=1 THEN
   PRINT "a ist eins"
   ELSEIF a=2 THEN
        PRINT "a ist zwei"
   ELSEIF a=3 THEN
        PRINT "a ist drei"
ELSE
   PRINT "a ist größer als drei"
ENDIF
```

Für solche Fälle gibt es auch noch die Anweisung SELECT ... CASE. Diese ist für mehrfache Entscheidungen der bessere Weg, weil sie übersichtlicher aufgebaut ist. Sie finden die Erklärung hierzu in der Befehlsübersicht.

Logische Abfragen

Logische Abfragen bringen nur zwei Antworten: WAHR und FALSCH. Diese heißen im Englischen TRUE (-1) und FALSE (0). Die Werte in den Klammern geben dabei den numerischen Wert dieser Konstanten an.
Die erste logische Abfrage haben wir in den vorhergehenden Beispielen schon des öfteren verwendet: IF a=1. Hier wird gefragt, ob (englisch: IF)

der Ausdruck a=1 WAHR (englisch: TRUE) ist. Nun wissen wir noch nicht, welchen Wert a hat, denn wir kennen nur die Abfrage, die Zuweisung der Variablen fehlt jedoch noch:

```
a=2
IF a=1 THEN
   PRINT "a ist eins"
ELSE
   PRINT "a ist nicht eins"
END IF
```

Welche Ausgabe wird dieses Programm liefern? Die Frage ob "a GLEICH 1" ist wird mit FALSCH (FALSE) beantwortet. Damit ist der Ausdruck IF a=1 falsch, denn a ist in diesem Fall 2. Das Programm überspringt also die nächste Zeile und kommt zu dem Befehl ELSE. Da die Frage falsch (FALSE) war, wird die andere Möglichkeit (die nach ELSE folgende Zeile) ausgegeben: a ist nicht eins.

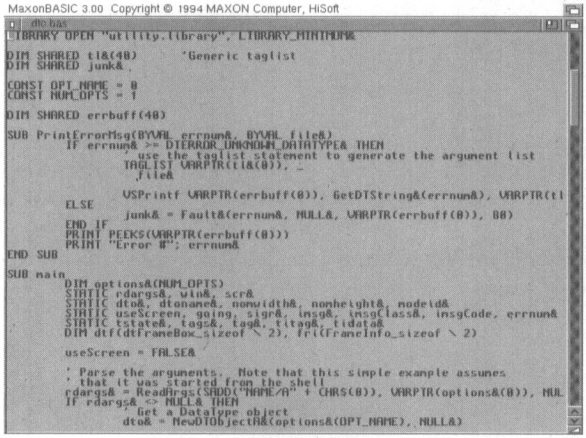
Es gibt noch mehrere logische Abfragen:

=	GLEICH
>	GRÖSSER
<	KLEINER
<>	NICHT GLEICH
>=	GRÖSSER ODER GLEICH
<=	KLEINER ODER GLEICH

Eine gute Einführung in die Programmierung sind natürlich die Beispielprogramme auf der Diskette. Besonders empfehlenswert für den Einstieg sind die Programme PLANTS.BAS, HANOI.BAS, LYAPUNOV.BAS, ASLTEST.BAS.
Sie finden diese Beispiele im Verzeichnis EXAMPLES/EXAMPLES in Ihrem MaxonBASIC-Ordner.

Der Editor

Der Editor ist das Herzstück von MaxonBASIC. Durch seine vollständige Einbindung in die Umgebung des AMIGA ist er sehr einfach zu bedienen. Aus ihm heraus können alle Aktionen auf intuitive und interaktive Art durchgeführt werden.
Der Editor bietet mehrere Textfenster, in denen verschiedene BASIC-Texte gleichzeitig bearbeitet werden können. Aktionen wie Texte eingeben, laden, speichern, drucken, Textstellen suchen und ersetzen, Blöcke löschen, ausschneiden und einfügen, können hier vorgenommen werden. Auch das Definieren und Benutzen von Makros wird vom MaxonBASIC-Editor unterstützt.
Der MaxonBASIC-Editor wird in zwei Versionen geliefert. Die erste Version ist für die Betriebssystem OS 2.x und OS 3.x geeignet. Die zweite Version ist für die Version 1.3 ausgelegt, bietet aber die gleichen Funktionen und bildet das Aussehen der OS 2.x-Version nach. Bei der Installation wird automatisch die richtige Version des Editors eingerichtet.



Der MaxonBASIC-Editor mit geladenem Quelltext

Starten des Editors

Der einfachste Weg, den MaxonBASIC-Editor zu starten ist ein Doppelklick auf das entsprechende Icon auf der Workbench. Der Editor kann jedoch auch vom CLI aus aufgerufen werden, wobei eine Liste der zu ladenden BASIC-Dateien angegeben werden kann.
Beim Start sucht der Editor nach der Datei MBASIC.prefs, und zwar zuerst im aktuellen Verzeichnis, dann im MaxonBASIC-Verzeichnis und zuletzt in ENV:MBASIC. Wird die Datei nicht gefunden, so werden Standardwerte verwendet.

Bedienung des Editors

Der Editor ist vollständig gemäß den Amiga-Richtlinien entwickelt worden. Dies bedeutet, daß er sich gemäß den Standards des Betriebssystems OS 2.0 oder höher verhält. Wenn Sie mit den prinzipiellen Funktionen der Menüs, Dialogfelder, Fenstern, Dateiauswahlfeldern usw. noch nicht vertraut sind, so sollten Sie zuerst das Handbuch, das Sie zu Ihrem AMIGA mitgeliefert bekamen, durcharbeiten.

Die Menüs des Editors

Das Menü PROJEKT

NEU
Wenn Sie NEU wählen, wird ein neues Textfenster mit dem Namen "Unbenannt 1" geöffnet. Wenn diese Aktion nicht gelingt, dann ist Ihr Speicher verbraucht. Schließen Sie nicht benutzte Textfenster oder Programme und versuchen Sie es erneut.

LADEN ...
Es öffnet sich ein Dateiauswahlfenster, in dem Sie die gewünschte Datei auswählen können. Wenn das aktuelle Textfenster schon einen Text enthält wird ein neues Fenster geöffnet.

ÖFFNEN ...
Es öffnet sich ein Dateiauswahlfenster, in dem Sie die gewünschte Datei auswählen können. Diese wird dann in das aktuelle Fenster geladen. Falls hier schon ein Text vorhanden ist, bekommen Sie eine entsprechende Meldung.

SPEICHERN
Die Datei im aktuellen Textfenster wird unter ihrem Namen abgespeichert.

SPEICHERN ALS ...
Es öffnet sich ein Dateiauswahlfenster, in dem Sie den Namen eingeben können, unter dem die Datei des aktuellen Textfensters gespeichert werden soll.

ÄNDERUNGEN SPEICHERN
Hier werden die aktuellen Texte aller Fenster gespeichert.

ZULETZT GESPEICHERT
Lädt die zuletzt gespeicherte Version des Textes im aktuellen Fenster. Bevor diese Aktion durchgeführt wird, erhalten Sie ein Warnung.

DRUCKEN
Druckt den Text des aktuellen Fensters auf dem Standard-Druckerausgang (PRT:) aus.

DRUCKEN ALS ...
Es öffnet sich ein Dialogfeld, in dem Sie die Anzahl der Kopien und den Druckerausgang angeben können. Die Option "Zeilennummern" gibt vor jeder Druckzeile eine Nummer aus, mit "Tabs erweitern" werden die Tabulatoren des Textes in Leerzeichen gewandelt.

ÜBER ...
Es erscheint ein Dialogfenster mit Angaben über die Rechte an MB, der Name des aktuellen Textes und dessen Zeilenlänge, Zeichenanzahl und das letzte Änderungsdatum.

BEENDEN
Beendet MB nach einer Sicherheitsabfrage (falls ein veränderter Text vorhanden ist).

Das Menü EDITIEREN

AUSSCHNEIDEN
Schneidet den markierten Text aus und legt ihn in einem Puffer ab. Der Text wird dabei gelöscht, kann jedoch mit EINFÜGEN wiederhergestellt werden.

KOPIEREN
Kopiert den markierten Text in einen Puffer.

EINFÜGEN
Fügt den Textteil aus dem Puffer an die aktuelle Cursorposition.

LÖSCHEN
Löscht den markierten Text.

ZEILE WIEDERHERSTELLEN
Stellt den ursprünglichen Zustand der aktuellen Zeile wieder her, solange man sie nicht verlassen hat.

BLOCK MARKIEREN
Nach Anwählen dieser Funktion kann mit den Cursortasten ein Block markiert werden.

UTILITY/EINRÜCKEN
Rückt die markierten Zeilen um eine Tabulatorposition nach rechts.

UTILITY/AUSRÜCKEN
Rückt die markierten Zeilen um eine Tabulatorposition nach links.

UTILITY/NUR GROSSB.
Wandelt alle Buchstaben des markierten Textes in Großbuchstaben um.

UTILITY/NUR KLEINB.
Wandelt alle Buchstaben des markierten Textes in Kleinbuchstaben um.

UTILITY/SCHLÜSSELWORT
Diese Funktion ist sehr praktisch, denn sie setzt alle BASIC-Schlüsselworte auf Großbuchstaben um. Variablennamen werden in Kleinbuchstaben gewandelt. Alle Texte in Anführungszeichen (Zeichenketten) sind von dieser Änderung ausgeschlossen.

DATEI EINFÜGEN ...
Fügt eine Datei an der aktuellen Cursorposition ein.

BLOCK SPEICHERN ...
Speichert den markierten Textbereich (Block) in einer Datei ab.

Das Menü SUCHEN

SUCHEN
Hiermit kann ein Wort oder ein Textteil gesucht werden.

SUCHEN & ERSETZEN
Neben dem Suchbegriff kann hier auch angegeben werden, womit er ersetzt werden soll.

SUCHEN/NÄCHSTEN
Sucht die nächste Textstelle, die mit dem Suchbegriff übereinstimmt.

SUCHEN/VORHERIGER
Sucht die vorangegangene Textstelle, die mit dem Suchbegriff übereinstimmt.

SUCHEN/ERSETZEN
Ersetzt den gefundenen Suchbegriff durch den Ersatztext.

GEHE ZU ZEILE
Springt direkt zu einer bestimmten Zeile.

GEHE ZUR LETZTEN ÄNDERUNG
Geht zu der Textstelle, die zuletzt geändert wurde.

STRUKTURTEST
Testet die Struktur des aktuellen BASIC-Programms. Primär werden die Verschachtelungen überprüft. Zum Testen muß man auf einem entsprechenden Schlüsselwort stehen.

GEHE ZU LESEZEICHEN
Springt zu einem bestimmten Lesezeichen.

SETZE LESEZEICHEN
Setzt ein Lesezeichen.

SORTIERE LESEZEICHEN
Sortiert die Lesezeichen nach aufsteigenden Zeilennummern.

Das Menü FENSTER

Diese Menüpunkte bestimmen die Anordnung und Auswahl der Fenster und sind alle selbsterklärend.

Das Menü PROGRAMM

STARTEN
Leitet die Phasen Tokenisierung, Compilierung und das Linken ein und startet das Programm (wenn kein Fehler aufgetreten ist!).

UNTERBRECHEN
Wenn ein Programm gestartet wurde und die Option "Abbruch Kontrolle" eingestellt wurde, dann kann es hiermit angehalten und abgebrochen werden.

FEHLER
Hiermit kann man im Editor die gemeldeten Fehler anspringen.

COMPILIEREN
Startet das Compilieren des aktuellen BASIC-Textes.

MAKE
Hier werden nur die Teile des Programms neu tokenisiert und compiliert, die verändert wurden.

PRÜFEN
Prüft die Syntax des aktuellen BASIC-Textes.

TOKENISIEREN
Bereitet den aktuellen BASIC-Text für das Compilieren vor. Dabei werden die BASIC-Befehle in vom Compiler verständliche Wörter (Tokens) übersetzt.

WICHTIG: Es bietet sich an, größere Programmteile, die mit INCLUDE eingebunden werden, zu tokenisieren und abzuspeichern. Bei weiteren Compilerläufen werden diese dann nur noch eingeladen und der Vorgang des Tokenisierens entfällt. Wichtig ist, daß dann die Zeilen mit INCLUDE aus dem BASIC-Text entfernt und die tokenisierte

Datei im Menü EINSTELLUNGEN unter dem Punkt COMPILER / DATEIEN in der Zeile TOKENS eingetragen werden müssen.

DEBUGGEN
Der Debugger dient der Fehlersuche. Er wird mit diesem Menüpunkt gestartet. Die Anleitung für den Debugger befindet sich auf der Diskette. Die Benutzung des Debuggers ist vor allem für Profis gedacht, denn hier wird direkt mit dem Betriebssystem gearbeitet. Für Einsteiger gibt es in MB einfachere Möglichkeiten um Fehler abzufangen, aufzudecken und zu beheben.

ARGUMENTE
Diese Funktion erlaubt die Eingabe von Werten, die dem Programm als Parameter beim Start übergeben werden.

AUSFÜHREN
Mit dieser Funktion wird das zuvor compilierte Programm gestartet. Sie haben die Möglichkeit, Parameter (Argumente) zu übergeben.

Das Menü MAKRO

MAKRO AUFNEHMEN
Startet die Aufnahme eines Makros

MAKRO BEENDEN
Beendet die Aufnahme eines Makros.

MAKRO STARTEN
Führt ein Makro aus.

MAKRO WIEDERHOLEN
Wiederholt die Ausführung des Makros.

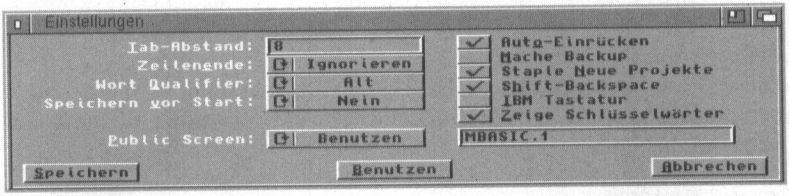
Das Menü EINSTELLUNGEN

AUF DATENTRÄGER?
Wenn diese Option angewählt ist, wird das BASIC-Programm auf den angegebenen Datenträger geschrieben.

ICON ERZEUGEN?
Ist diese Option angewählt, wird beim Speichern jedes BASIC-Textes ein Icon angelegt. In der Infostruktur des Icons werden dabei wichtige Einstellungen abgespeichert, die nur für diesen Text gültig sind.

EINSTELLUNGEN

Hier können die folgenden Einstellungen für den Editor vorgenommen werden.



Tab-Abstand

Die Standardweite beträgt 8 Zeichen, sie kann jedoch von 1-20 variiert werden.

Zeilenende

Bestimmt das Verhalten am Zeilenende

Speichern vor Start

Speichert den BASIC-Text vor dem Starten des Programms.

Public Screen

Folgende Möglichkeiten stehen zur Auswahl: Erzeugen (erzeugt einen neuen Bildschirm mit dem angegebenen Namen), NEU (erzeugt einen neuen Bildschirm), BENUTZEN (benutzt den vorhandenen Bildschirm)

Auto. Einrücken

Wenn diese Option eingestellt ist, wird das Einrücken eines Listings eingehalten. Dies bedeutet, daß wenn Sie in einer eingerückten Zeile schreiben und <Return> drücken, der Cursor nicht in die erste Spalte zurückspringt, sondern bei der Einrückung der vorangegangenen Zeile stehenbleibt.

Mache Backup

Erstellt bei jedem Speichern des BASIC-Textes eine Sicherheitskopie mit der Endung "BAK".

Staple Neue Projekte

Jedes neu geöffnete Projekt-Fenster wird direkt unter der Titelleiste des vorhergehenden Fensters angeordnet. So behält man besser den Überblick bei mehreren geöffneten Fenstern und kann sie direkt anwählen.

IBM-Tastatur
Diese Option belegt die Tasten des 10er-Blocks gemäß der Funktion NUMLOCK. D.h., daß die Tasten mit den Bewegungen im Text belegt sind (die Bedeutung steht bei den meisten AMIGA-Tastaturen).

Zeige Schlüsselwörter
Diese Option bewirkt, daß die BASIC-Schlüsselworte in Großbuchstaben umgesetzt werden. Die Funktion tritt in Aktion, sobald Sie die Zeile verlassen haben.

WERKZEUGE
Hinter diesem Menüeintrag verbergen sich die Optionen zum Laden des Compilers und des Debuggers. Um Speicher zu sparen können beide erst dann geladen werden, wenn man sie benötigt. Wer genug Speicher zur Verfügung hat, kann auch bestimmen, daß beide Programme sofort geladen werden.
Mit diesem Menüpunkt kann man den Speicher, den Compiler und Debugger belegen, auch wieder freigeben.

ANZEIGE
Mit diesem Menü können die Zeichensätze für den Text und den Bildschirm bestimmt werden. Ebenso läßt sich hier die Auflösung des Editor-Bildschirms einstellen.

COMPILER
Hier werden die Einstellungen für den Compiler vorgenommen. Genauere Angaben finden Sie im nächsten Kapitel (siehe Compiler).

EINSTELLUNGEN LADEN ...
Wer mehrere verschiedene Einstellungen abgespeichert hat, kann hier ein bestimmtes Set wieder laden.

EINSTELLUNGEN SPEICHERN
Speichert die Einstellungen unter dem Standardnamen ab.

EINSTELLUNGEN SPEICHERN ALS ...
Speichert die Einstellungen unter einem wählbaren Namen ab.

Der Compiler

MB ist ein BASIC-Compiler im Gegensatz zu einem BASIC-Interpreter (wie z.B. AmigaBASIC). Ein Interpreter liest eine Textzeile, interpretiert den Inhalt und führt ihn aus. Ein Compiler übersetzt den BASIC-Quelltext in Assemblercode, der dann in Maschinencode übersetzt wird. Der Maschinencode wird vom Computer direkt verstanden und daher viel schneller ausgeführt. Mit MB können Sie daher sehr schnelle Programme erzeugen und müssen dabei nicht auf den Komfort und die Einfachheit von BASIC verzichten.

Es gibt zwei Möglichkeiten, ein Programm zu compilieren und danach zu starten. Die erste und einfachere geschieht über einen Menüaufruf im Editor. Die andere geht über einen Aufruf vom CLI aus.

Das PROGRAMM-Menü

Alle Aktionen des Compilier-Vorgangs sind unter dem Menüpunkt PROGRAMM zusammengefaßt. Dies sind im einzelnen folgende Punkte:

STARTEN
Leitet die Phasen Tokenisierung, Compilierung und das Linken ein und startet das Programm (wenn kein Fehler aufgetreten ist!). Beim Starten wird zudem der unter ARGUMENTE angegebene Text an das Programm übergeben.

ACHTUNG: Bitte denken Sie immer daran, daß wenn Ihr Programm abstürzt, der Quelltext aus dem Editor gelöscht werden kann. Wenn Sie ihn vorher nicht von Hand gespeichert oder die Option "Speichern vor Start" im Menü "Einstellungen" angewählt haben, dann kann Ihr Quelltext unwiderruflich verlorengehen.

UNTERBRECHEN
Wenn ein Programm gestartet wurde und die Option "Abbruch-Kontrolle" eingestellt wurde, dann kann es hiermit angehalten und abgebrochen werden.

FEHLER
Hiermit kann man im Editor die gemeldeten Fehler anspringen.

COMPILIEREN
Startet das Compilieren des aktuellen BASIC-Textes.

MAKE
Hier werden nur die Teile des Programms neu tokenisiert und compiliert, die verändert wurden.

PRÜFEN
Prüft die Syntax des aktuellen BASIC-Textes.

TOKENISIEREN
Bereitet den aktuellen BASIC-Text für das Compilieren vor. Dabei werden die BASIC-Befehle in vom Compiler verständliche Wörter (Tokens) übersetzt.

WICHTIG: Es bietet sich an, größere Programmteile, die mit INCLUDE eingebunden werden, zu tokenisieren und abzuspeichern. Bei weiteren Compilerläufen werden diese dann nur noch eingeladen und der Vorgang des Tokenisierens entfällt. Wichtig ist, daß dann die Zeilen mit INCLUDE aus dem BASIC-Text entfernt und die tokenisierte Datei im Menü EINSTELLUNGEN unter dem Punkt COMPILER / DATEIEN in der Zeile TOKENS eingetragen werden müssen.

DEBUGGEN
Der Debugger dient der Fehlersuche. Er wird mit diesem Menüpunkt gestartet. Die Anleitung für den Debugger befindet sich auf der Diskette. Die Benutzung des Debuggers ist vor allem für Profis gedacht, denn hier wird direkt mit dem Betriebssystem gearbeitet. Für Einsteiger gibt es in MB einfachere Möglichkeiten um Fehler abzufangen, aufzudecken und zu beheben (siehe TRON/TROFF).

ARGUMENTE
Diese Funktion erlaubt die Eingabe von Werten, die dem Programm als Parameter beim Start übergeben werden.

AUSFÜHREN
Mit dieser Funktion wird das zuvor compilierte Programm gestartet. Sie haben die Möglichkeit, Parameter (Argumente) zu übergeben.

Das Compilieren

Das Compilieren unterteilt sich in die Phasen Tokenisieren, Parsen und Code-Generierung. Wenn ein Fehler bei einem der Vorgänge auftritt, so erscheint eine entsprechende Meldung. Sie können jedoch das Compilieren bis zum Ende weiterlaufen lassen, um zu erfahren, ob noch weitere Fehler vorhanden sind.
Drücken Sie OK, wenn Sie auch die weiteren Fehlermeldungen lesen wollen. Wenn Sie WEITER anwählen, dann läuft der Vorgang zwar auch weiter, die Fehler werden jedoch nicht mehr angezeigt. Dennoch können Sie sie danach im Editor nacheinander anspringen (NÄCHSTER / VORHERIGER FEHLER).

Compileroptionen

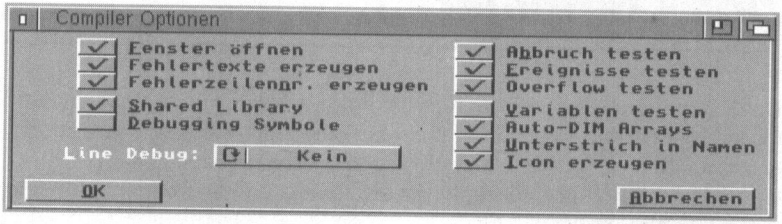
Es gibt prinzipiell zwei Möglichkeiten, Compileroptionen einzugeben. Der einfache Weg ist über das Editor-Menü EINSTELLUNGEN / COMPILER. Der zweite Weg führt über "REM $"-Eingaben im BASIC-Quelltext. Diese Methode ist notwendig, wenn man den Compiler vom CLI aus aufruft.
Ein Beispiel soll die Benutzung verdeutlichen:

```
REM $LINES
```

Dieser Befehl entspricht dem Punkt "Fehlerzeilennr. erzeugen" im Auswahlfeld der Compiler-Optionen. Wird ein solcher Befehl im Quelltext eingetragen, so überschreibt er die Einstellungen in den Compiler-Optionen.
Die meisten Befehle können mit "NO" wieder deselektiert (abgewählt) werden.

```
REM $NOLINES
```

Es folgt die Aufstellung der Einstellungen des Auswahlfeldes **Compiler-Optionen**. In Klammern wird der Befehl angegeben, der im Quelltext verwendet werden muß.



"Fenster öffnen" (WINDOW)
Öffnet das Standard-Ausgabefenster.

"Fehlertexte erzeugen" (ERRORS)
Erzeugt außer der sonst üblichen Fehlernummer den dazugehörigen Fehlertext.

"Fehlerzeilnenr. erzeugen" (LINES)
Erzeugt die Zeilennummer der fehlerhaften Zeile.

"Shared Library" (LIBRARY)
Wenn diese Option benutzt wird, werden kürzere Programme erzeugt, die jedoch die "mbasic3.library" bzw. "hbasic2.library" im Pfad "LIBS:" benötigt. Wenn die erzeugten Programme weitergegeben werden sollen, dann ist es besser, diese Option auszuschalten, ansonsten ist es unbedingt notwendig die Library mitzuliefern.

"Debugging Symbole" (DEBUG)
Ist diese Option gesetzt, werden Symbole für den Debugger erzeugt, damit dieser die Einsprünge für Unterprogramme und Funktionen finden kann.

"Line Debug Standard" (LINEDEBUG)
"Line Debug komprimiert" (HCLNDEBUG)
Diese Funktion erzeugt Informationen für den Debugger, damit dieser die Verbindung zur richtigen Zeilennummer im Quelltext herstellen kann. Die komprimierte Version benötigt nur 2 Byte pro Zeile, wird jedoch nur vom MonAm-Debugger verstanden. Die "Standard"-Version benötigt 8 Byte pro Zeile und kann auch von CodeProbe und anderen Debuggern bearbeitet werden.

"Abbruch testen" (BREAK)
Ist diese Option gesetzt, dann kann das Programm immer mit <CTRL>+<C> angehalten werden. Ist sie nicht gesetzt, so kann es nur

bei den Aktionen PRINT, INPUT und INKEY$ abgebrochen werden. Die Funktion kann innerhalb des Quelltextes an- und abgeschaltet werden.

"Ereignisse testen" (EVENT)
Wenn Sie den Befehl "ON .. xxxx" verwenden, dann müssen die diese Option einschalten um die Überprüfung zu gewährleisten. Ansonsten sollten Sie sie abschalten, denn diese spart Zeit und das Programm bleibt kürzer.

"Overflow testen" (OVERFLOW)
Diese Funktion testet, ob eine Variable über ihren vorgegebenen Wert hinausgeht. Für mathematische und einige E/A-Operationen werden 2 zusätzliche Byte verwendet. Hierdurch nimmt die Geschwindigkeit leicht ab.

WICHTIG: Sie sollten diese Funktion nur abschalten, wenn Sie sicher sind, daß Ihr Programm diesbezüglich einwandfrei funktioniert. Einen "Division durch Null"-Fehler fängt diese Funktion nicht ab!

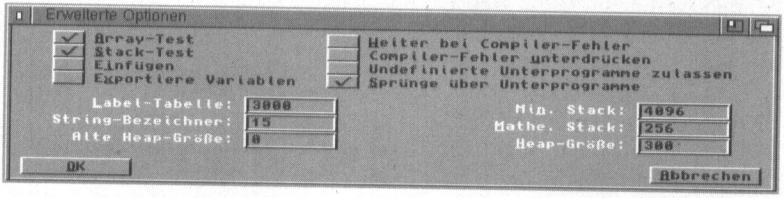
"Variablen testen" (VARCHECKS)
Wenn eine Feldvariable z.B. i(x), die mit DIM i(10) festgelegt wurde, mit i(11) angesprochen wird, dann tritt der Variablentest in Kraft, um Zugriffe auf nicht zulässige Speicherstellen zu verhindern. Diese Funktion ist sehr praktisch, denn dieser Fehler tritt häufig auf und kann zu sehr fatalen Fehlern führen.

"Auto-DIM Arrays" (AUTODIM)
Diese Funktion führt für jedes Variablenfeld automatisch eine Dimensionierung auf 10 durch. Dies ist in einigen BASIC-Dialekten der Standard, allerdings sollte man jedes Variablenfeld eigenhändig dimensionieren um die Kontrolle darüber zu behalten.

"Unterstrich im Namen" (UNDERLINES)
Diese Funktion wurde aus Kompatibilitätsgründen zu Microsoft BASIC eingeführt. Ist sie angeschaltet, so können Unterstriche in Variablen- und Prozedurnamen verwendet werden.

"Icon erzeugen" (ADDICON)
Zu jedem Programm wird ein Icon ".info-Datei" erzeugt.

Die folgende Einstellungen können bei "**Erweiterte Optionen**" vorgenommen werden.



"Array-Test" (ARRAY)
Ist diese Option gesetzt, so wird jeder Zugriff auf ein Variablenfeld (Array) getestet und, falls es nicht richtig dimensioniert wurde, eine Fehlermeldung ausgegeben. Wenn die Option nicht gesetzt wird, dann führen solche Zugriffe auf fremde Speicherbereiche häufig zu fatalen Fehlern, deren Ursache nur schwer zu finden ist.

"Stack-Test" (STACK)
Diese Funktion testet vor jedem RETURN, ob es auch ein dazugehöriges GOSUB gibt. Ist sie ausgeschaltet, dann können solche Programme unvorhersehbare Fehler produzieren.

"Exportiere Variablen" (EXPORTS)
Diese Funktion ist dann notwendig, wenn Assembler-Texte dazugelinkt werden sollen. Dann werden alle globalen Variablen, Unterprogramme und Funktionen exportiert.

"Weiter bei Compiler-Fehler" (BATCH)
Normalerweise stoppt der Compiler beim ersten Fehler und gibt dem Anwender die Möglichkeit, diesen sofort zu beheben. Möchte man sich jedoch einen Eindruck aller auftretenden Fehler verschaffen, so kann man diese Funktion einschalten.

"Compiler-Fehler unterdrücken" (NOWARN)
Ist diese Option gesetzt, werden keinerlei Fehlermeldungen ausgegeben.

"Undefinierte Unterprogramme zulassen" (UNDEFSUBS)
Wenn diese Option gewählt wurde, dann gibt es keine Warnung wenn das Programm Unterprogramme aufruft, die nicht definiert sind.

"Sprünge über Unterprogramme" (NOJUMPS)
Normalerweise fügt das BASIC vor jedem Unterprogramm einen Sprung auf die nächste Zeile hinter dem Unterprogrammende ein, damit die Unterprogramme nicht "durchlaufen" bzw. ausgeführt werden. Wenn Sie diese Funktion abschalten, dann sollten Sie sicherstellen, daß die

Unterprogramme nicht versehentlich vom Programm "durchlaufen" werden. Dies kann dann schnell passieren, wenn Sie am Anfang des Textes stehen.

"Label-Tabelle" (LABELS)
Der eingestellte Wert gibt die Anzahl der Labels an, die der Code-generator verwenden kann. Jeder Labeleintrag benötigt 6 Byte, wird der eingestellte Wert überschritten erscheint ein "Label Tabelle voll"-Fehler.

"String-Bezeichner" (STRINGS)
Dieser Wert gibt die maximale Anzahl der temporären Strings an. Wenn die Fehlermeldung "String zu komplex" erscheint, dann sollte dieser Wert erhöht werden.

"Alte Heap-Größe" (KEEP)
Der Wert gibt die Größe des Heaps in KB an. Der Wert kann bei einem CLI-Aufruf des Programms mit dem Parameter HEAP=size vergrößert werden.

"Min. Stack" (MINSTACK)
Auf dem Stack werden bei GOSUBs, Funktions- und Unterprogramm-aufrufen die Rücksprungadressen abgelegt. Der Stack muß mindestens 200 Byte betragen. Standardmäßig werden 4096 Byte reserviert. Bei stark rekursiven Programmen kann es notwendig werden, diesen Wert zu erhöhen.

"Mathe. Stack" (MATHSTACK)
Auch bei mathematischen Funktionen wird ein Stack benötigt. Er muß mindestens 32 Byte groß sein. Standardmäßig sind es 256 Bytes. Er muß aber nur bei sehr komplizierten numerischen Ausdrücken erhöht werden.

"Heap-Größe" (HEAPDYNAMIC)
In einem Multitasking-System ist es keine gute Sitte, sich schon beim Programmstart möglichst viel Speicher zu reservieren. Daher wurde eine dynamische Heap-Verwaltung eingeführt, die sich immer nur einen Teil des Speichers nimmt.

WICHTIG: Der Nachteil dieser Methode ist, daß Sie zur Fragmentierung des Speichers führt, also zu einer Aufteilung in kleine, nicht zusammenhängende Speicherstücke. Programme, die ein großes zusammenhängendes Speicherstück benötigen, erhalten dann eine Absage, obwohl die Gesamtmenge des Speichers noch größer ist.

Folgende Einstellungen können bei "**Datei-Optionen**" vorgenommen werden:

"Compile" (FROM)
Hier kann die zu compilierende Datei angegeben werden. Ist das Feld leer, wird automatisch die Datei des aktuellen Fensters compiliert.

"Ausgabe nach" (TO)
Normalerweise wird beim Speichern des ausführbaren Programms der Name des aktuellen Quelltextes ohne die Endung ".BAS" verwendet. Wollen Sie einen anderen Namen, so können Sie dies hier einstellen.

"Token-Datei" (TOKENS)
Hier können Sie die tokenisierte Datei angeben, die beim Compilieren verwendet werden soll.

"Include-Pfad" (INCPATH)
Hier können Sie den Pfad angeben in dem der Compiler nach den Include- und Token-Dateien sucht. Normalerweise sucht der Compiler zuerst in dem Verzeichnis, das die System-BH-Dateien enthält, danach im aktuellen Verzeichnis und in dem mit INCPATH angegebenen. Falls alles fehlschlägt sucht er noch in "INCBAS:".

"Library-Pfad" (LIBPAHT)
Hier können Sie den Pfad angeben in dem der Compiler die BMAP-Dateien sucht. Normalerweise werden diese im BMAP-Verzeichnis innerhalb des MaxonBASIC-Verzeichnisses gesucht. Falls er sie dort nicht findet sucht er noch im LIBS-Verzeichnis.

Weitere Optionen, die nicht mit den Einstellern von MaxonBASIC vorgenommen werden können, sondern nur für den Einsatz mit dem CLI-Compiler verwendet werden können:

"Tokenise" (TOKENISE)
Diese Option veranlaßt den CLI-Compiler nur eine Tokenisierung des Quelltextes vorzunehmen und diesen abzuspeichern.

"Check Source Code" (CHECK)
Diese Option veranlaßt den CLI-Compiler den Quelltext zu compilieren, aber nicht abzuspeichern. Dies ermöglicht eine Kontrolle des Quelltextes nach Fehlern usw.

"Compile" (COMPILE)
Diese Option ist standardmäßig eingeschaltet. Der Quelltext wird normal compiliert.

"Remove reserved words" (DISABLE)
Mit dieser Option werden bestimmte Worte, die in MaxonBASIC als reservierte Worte definiert sind, ausgeblendet. Jeder BASIC-Dialekt definiert bestimmte Worte als reserviert (z.B. die BASIC-Befehle), d.h. daß diese Worte nicht als Variablen-, Prozedur- oder Funktionsnamen verwendet werden dürfen. Beim Konvertieren von Quelltexten anderer BASIC-Dialekte kann es jedoch vorkommen, daß dort Namen verwendet werden, die reservierten Worten entsprechen. Daher die Möglichkeit, bestimmte reservierte Worte auszublenden. Die entsprechenden Befehle von MaxonBASIC sind dann jedoch nicht mehr verfügbar, wohl aber wird das Programm laufen.

Der Programmteil

```
FOR loop=1 TO 10
NEXT loop
```

wird in MaxonBASIC zu einer Fehlermeldung führen. Mit dem Befehl

```
REM $DISABLE LOOP
```

wird dies ermöglicht.

WICHTIG: Verwenden Sie diese Option nur wenn Sie es sehr eilig mit der Umsetzung haben und entfernen Sie die Einschränkung so schnell wie möglich.

"Output Icons" (ICON)
Wird diese Option angegeben, so wird eine Icon-Datei (.info) erzeugt.

"Suppress compiler titles" (QUIET)
Diese Option unterdrückt alle Compiler-Ausgaben außer den Fehlermeldungen.

"Take options from file" (WITH)
Dies veranlaßt den CLI-Compiler dazu, die Optionen aus einer separaten Datei einzulesen.

Menüs der Compiler-Optionen

Während die Compiler-Optionen-Einsteller geöffnet sind, erhalten Sie mit der rechten Maustaste eine Reihe von Menüs. Die meisten Punkte sind dabei selbsterklärend. Nachfolgend die Optionen des Unterpunktes "Presets". Dies sind Voreinstellungen, die alle notwendigen Optionen in den vorangegangenen Menüs einstellen:

KOMPATIBILITÄT
Diese Option sollten Sie beim Compilieren alter AmigaBASIC-Programme einsetzen.

SICHERHEITSTESTS
Dieser Menüpunkt schaltet auf maximale Überprüfung des Quelltextes. Die Zeiten und die Programmgröße nehmen dabei zu.

DEBUGGING
Die Option nimmt alle Einstellungen für das Debuggen vor.

SCHNELLER CODE
Dies erzeugt den schnellstmöglichen Code.
Bitte beachten Sie, daß dabei möglicherweise Optionen ausgeschaltet werden, die von Ihrem Programm benötigt werden!

KOMPAKTER CODE
Dies erzeugt besonders kleine Programme.
Bitte beachten Sie, daß dabei möglicherweise Optionen ausgeschaltet werden, die von Ihrem Programm benötigt werden!

Benutzung des CLI-Compilers HBC

Zusätzlich zu der Verwendung des Compilers vom Editor aus, kann er auch über das CLI bzw. die SHELL verwendet werden. Der CLI-Compiler heißt HBC, der Aufruf lautet:

```
HBC quelltext [optionen]
```

Der "Quelltext" ist das BASIC-Programm mit oder ohne "BAS"-Endung. Bei "Optionen" werden einzelne oder eine Liste von Einstellungen (siehe oben) angegeben.

WICHTIG: Die Optionen sind beim CLI-Compiler standardmäßig ausgeschaltet !!!

Ein Beispiel:

```
HBC test
```

compiliert das Programm "test.bas", wobei alle Optionen ausgeschaltet sind!

```
HBC test overflow lines arrays
```

compiliert das selbe Programm, schaltet aber den "Overflow-Test", "Fehlerzeilennummern erzeugen" und "Array-Test" an.

Erstellung von CLI-Programmen

Wenn Sie ein Programm erzeugen wollen, daß in der CLI- oder SHELL-Umgebung abläuft, dann müssen Sie die Option "NOWINDOW" verwenden.
Wenn Ihr CLI-Programm mit einem Icon gestartet werden soll, dann können Sie automatisch ein Ein/Ausgabe-Fenster öffnen. Tragen Sie dazu im Tooltype des Icons folgende Zeile ein:

```
WINDOW=CON:0/0/640/200/Automatic Windows!
```

Tokenisieren

Beim Tokenisieren wird der aktuelle BASIC-Text für das Compilieren vorbereitet. Dabei werden die BASIC-Befehle in vom Compiler verständliche Wörter (Tokens) übersetzt.

Es bietet sich an, größere Programmteile, die nicht mehr verändert werden, zu tokenisieren und abzuspeichern. Bei weiteren Compilerläufen werden diese dann nur noch eingeladen und der Vorgang des Tokenisierens entfällt.

Wenn Sie in Ihrem Quelltext einen größeren Teil haben, den Sie voraussichtlich nicht mehr ändern müssen, dann ist dieser geeignet für eine Token-Datei. Dies könnte z.B. der Vorspann Ihres Programms sein, der die Systemincludes einfügt:

```
REM $INCLUDE Exec.bh
REM $INCLUDE Intuition.bh
REM $INCLUDE Graphics.bh
```
usw.

Schneiden Sie die Zeilen mit den "REM $INCLUDES xxx.bh"-Befehlen aus und schreiben Sie sie in eine neue Datei. Diese Datei wird nun tokenisiert. Wählen Sie im Menü PROGRAMM den Punkt "Tokenisieren". Der Vorgang beginnt und es wird in das gleiche Verzeichnis, in dem sich auch Ihr restlicher Quelltext des Programms befindet, eine Datei mit dem Namen des Quelltextes und der Endung ".t" geschrieben.

Wenn Sie nun diesen vor-tokenisierten Programmteil verwenden wollen, dann laden Sie wieder den Quelltext Ihres Hauptprogramms und löschen die Zeilen, die in der Token-Datei verwendet wurden. Dann müssen Sie den Namen der Token-Datei im Menü EINSTELLUNGEN/ COMPILER/DATEIEN ... unter dem Punkt "Token-Datei" angeben. Die entsprechende Datei befindet sich im allgemeinen im gleichen Verzeichnis in dem auch Ihr Programm abgespeichert wurde. Sie hat die Endung ".t".
Wählen Sie nun den Punkt "STARTEN" im Menü PROGRAMM. Sie sehen nun, wie die Token-Datei geladen wird. Danach folgen die gleichen Schritte, wie bei einem normalen Compilerlauf.

WICHTIG: Vergessen Sie nicht, den Teil, der in der Token-Datei enthalten ist, auch aus Ihrem Quelltext zu entfernen. Heben Sie aber auf jeden Fall den Quelltext der Token-Datei auf, denn sonst können Sie daran keine Änderungen mehr vornehmen.

Einschränkungen beim Tokenisieren:

- Die Token-Datei wird als erste Datei vom Compiler bearbeitet (geparsed).
- Die Token-Datei darf keine DATA-Befehle enthalten
- Es kann nur eine Token-Datei angegeben werden. Allerdings können Sie weitere Teile des Quelltextes zu einer bestehenden Token-Datei hinzufügen. Geben Sie dazu im Menü EINSTELLUNGEN/ COMPILER/DATEIEN ... unter dem Punkt "Token-Datei" die bestehende Datei an, laden Sie den neuen Quelltext in den Editor und wählen Sie TOKENISIEREN.
- Denken Sie daran, daß, fall Sie Compiler-Optionen in der Token-Datei angegeben haben, diese auch verwendet werden, auch wenn Ihr restlicher Hauptquelltext diese nicht einsetzt.

Maxon BASIC 3

Features

Compiler
- sehr großer Befehlsumfang
- 100%-HiSoft BASIC kompatibel
- weitgehend kompatibel zu Amiga-BASIC, Microsoft QuickBASIC u. a.
- professionelle Strukturelemente (Includes, Unterroutinen, Funktionen)
- sehr schneller Compiler
- vorcompilierte Includes für schnelle Compilierzeiten
- erzeugte Programme laufen auf allen Prozessoren (68000 - 68040)
- Multitasking-Unterstützung
- lauffähig auf allen Amiga-Betriebssystemen (OS 1.3 - OS 3.1) und allen Amiga-Modellen (A500 - A4000) mit mindestens 1MB Speicher und Diskettenlaufwerk
- deutschsprachige Programmversion

Editor
- komfortabler Mehrfenster-Editor
- auf der Workbench lauffähig
- auch für Grafikkarten geeignet

Bibliotheken
- Systemzugriffe auf alle Amiga-Funktionen (OS 1.3 bis OS 3.1)
- einfache Portierungsmöglichkeiten von C- und Pascal-Programmen

Debugger
- Debugger mit Anzeige des BASIC-Textes für die schnelle Fehlersuche

Beispielprogramme
- zur Programmierung von Gadtools, IFF-Laden- und -Speichern, Datatypes, Tasks, Sounds uvm.

Der professionelle BASIC-Compiler

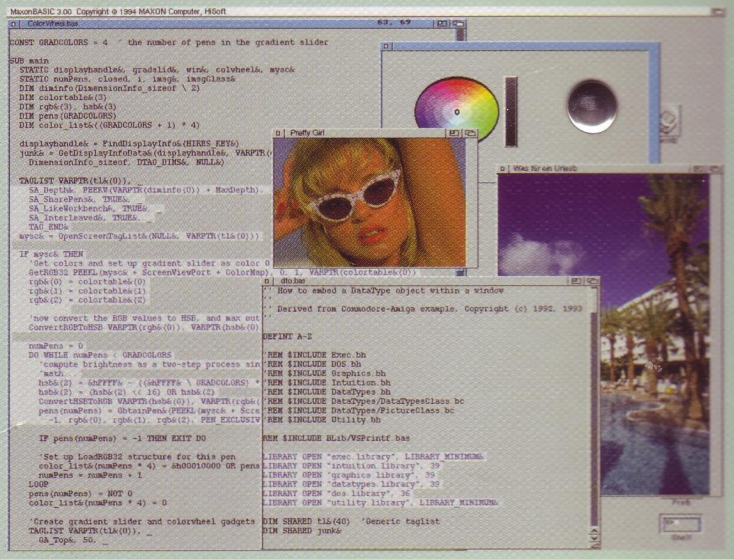
Mit dem neuen MaxonBASIC kann jeder die neuen Features des Amiga-Betriebssystems 3.1 schnell und sicher verwenden. Völlig systemkonform und mit viel Komfort präsentiert sich die Arbeitsumgebung und macht damit die Programmierung des Amiga sehr schnell und einfach. Moderne Strukturelemente und leistungsfähige Funktionen machen MaxonBASIC zu der geeigneten Sprache für professionelle Entwicklungen.

Einfache Handhabung

Für Einsteiger sind das einfach zu bedienende Komplettsystem und der umfangreiche BASIC-Dialekt die ideale Voraussetzung für den Start in die Amiga-Programmierung.

Großer Befehlsumfang

Der BASIC-Befehlsumfang ist sehr groß und weitgehend kompatibel zu den bekannten Dialekten HiSoft BASIC, AmigaBASIC, Microsoft QuickBASIC und anderen. Zudem werden die Funktionen aller Amiga-Betriebssysteme von 1.3 bis 3.1 unterstützt. Moderne Strukturelemente wie Includes, Unterroutinen und Funktionen, zusammen mit leistungsfähigen Befehlen erlauben ein professionelles Programmieren wie mit anderen Compilern (Pascal, C, Modula-2).



Editor

Herzstück des Systems ist ein sehr guter und einfach zu bedienender Editor, der völlig systemkonform gemäß den Richtlinien von Commodore entwickelt wurde. Von hier aus werden alle Funktionen des Compilersystems per Menü aufgerufen. Im Editor können mehrere Quelltexte geöffnet sein, zwischen denen Texte ausgetauscht werden können.

Compiler und Debugger

Der Compiler ist sehr schnell und erzeugt kompakte, sofort ausführbare Programme für alle Prozessortypen. Der symbolische Debugger und Disassembler unterstützt die Fehlersuche in Programmen und zeigt dabei auch den BASIC-Quelltext an.

MaxonBASIC entstand in Zusammenarbeit mit der englischen Firma HiSoft und besteht aus einem leistungsfähigen Editor, einem sehr schnellen Compiler und einem Debugger mit Quelltextanzeige.

Geliefert wird MaxonBASIC als deutschsprachige Programmversion mit vielen Beispielen und allen notwendigen Dateien für die Betriebssystemprogrammierung.

Die Systemvoraussetzungen sind ein AMIGA (A500 - A4000) mit mindestens 1MB RAM, Betriebssystem 1.3 bis 3.1 und ein Diskettenlaufwerk.

AMIGA®

4 010509 411104

MAXON Computer GmbH • Industriestr. 26
D-65760 Eschborn • Telefon 06196/481811
Irrtümer und Änderungen vorbehalten!

MAXON computer